VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Rychlá fotorealistická vizualizace komplexních modelů

Fast Photorealistic Visualization of Complex Models

2018 Bc. Jakub Bílý

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Jakub Bílý**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Rychlá fotorealistická vizualizace komplexních modelů
Fast Photorealistic Visualization of Complex Models

Jazyk vypracování: čeština

Zásady pro vypracování:

Cílem práce je navrhnout a naimplementovat knihovnu pro rychlou fotorealistickou vizualizaci komplexních modelů reálných objektů pomocí Vulkan API. Získané obrázky budou následně použity pro generování vzorů možných poloh zájmových objektů. Součástí implementace bude také výpočet vhodných deskriptorů zobrazovaných objektů pro jejich následné vyhledávání v reálném obraze. Implementace bude provedena v jazyce C++ za dodržení Google C++ Style.

1. Seznamte se s možnostmi Vulkan API pro generování fotorealistických snímků zadaných modelů skrze programové rozhraní.
2. Vytvořte knihovnu, která bude umožňovat skrze toto rozhraní vizualizovat zvolený model při různých pohledech kamery.
3. U vygenerovaných snímků vypočítejte vhodné deskriptory a porovnejte je s deskriptory vypočtenými nad korespondujícími reálnými snímky téhož objektu.
4. Zvolené postupy a dosažené výsledky pečlivě popište v textové části práce.

Seznam doporučené odborné literatury:

[1] Khronos Group. Vulkan API Specifications.
[2] Karis, B. Real Shading in Unreal Engine 4. Siggraph 2013.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Tomáš Fabián, Ph.D.**

Datum zadání: 01.09.2016
Datum odevzdání: 30.04.2018

doc. Ing. Jan Platoš, Ph.D.
vedoucí katedry

prof. Ing. Pavel Brandštetter, CSc.
děkan fakulty

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 25. dubna 2018 ……..

Rád bych na tomto místě poděkoval svému vedoucímu diplomové práce Ing. Tomáši Fabiánovi, Ph.D. za odborné vedení a cenné rady.

Abstrakt

Hlavním tématem této diplomové práce je vytvoření aplikace umožňující rychlou vizualizaci komplexních modelů reálných objektů pomocí Vulkan API při různých pohledech kamery. Získané obrázky jsou následně použity pro generování vzorů možných poloh zájmových objektů. Součástí této diplomové práce je také výpočet klíčových bodů pomocí vhodných detektorů a deskriptorů zobrazovaných objektů pro jejich následné vyhledávání v reálném obraze za pomocí knihovny OpenCV a algoritmů ULATCH a TILDE. Implementace je provedena v jazyce C++.

Klíčová slova: vizualizace modelu, Vulkan API, 3D model, OpenCV, tinyobjloader, stbi-image, feature detektor, feature deskriptor, ULATCH, TILDE, PBR, LaTeX

Abstract

The main theme of this diploma thesis is the creation of an application enabling rapid visualization of complex models of real objects using Vulkan API in various camera views. The images obtained are then used to generate patterns of possible locations of objects of interest. Part of this diploma thesis is also the keypoints calculation with help of suitable detectors and descriptors of displayed objects for their subsequent search in real images using the OpenCV library and the ULATCH and TILDE algorithms. Implementation is done in C ++.

Key Words: visualization of model, Vulkan API, 3D model, OpenCV, tinyobjloader, stbi-image, feature detector, feature descriptor, ULATCH, TILDE, PBR LaTeX

Obsah

Seznam použitých zkratek a symbolů

Vertex buffer	–	buffer obsahující vrcholy modelu
Index buffer	–	buffer obsahující indexy na unikátní vrcholy modelu
MVP	–	model-view-projection matice
Color blending	–	prolnutí barev v pixelech
GLM	–	označení pro knihovnu OpenGL Mathematics
HPC	–	high performance computing neboli vysoce výkonné výpočty například na superpočítačích
PBR	–	physically based rendering neboli snaha o fotorealistické renderování objektů
API	–	application programmable interface neboli rozhraní pro programování aplikací
OBJ	–	jeden z formátů běžných pro export 3D modelů obsahující geometrii, normály atd.
MTL	–	běžný formát přidružený k OBJ formátu 3D modelů obsahující informace o materiálech a texturách
GPU	–	graphical processing unit neboli grafická karta počítače
GLSL	–	OpenGL Shading Language neboli programovací jazyk shaderů pro OpenGL
SPIR-V	–	programovací jazyk shaderů pro Vulkan API
GUI	–	Graphical User Interface neboli grafické prostředí aplikace

Seznam obrázků

Seznam tabulek

Seznam výpisů zdrojového kódu

1 Úvod

Hlavním cílem této diplomové práce je vytvoření samostatné aplikace umožňující rychlou vizualizaci komplexních modelů reálných objektů ve formátu .obj pomocí Vulkan API při různých pohledech kamery. Tomuto úkolu nejprve předchází rešerše o samotném Vulkan API a jeho schopnostech. Dalším cílem této práce je výpočet klíčových bodů zobrazovaných objektů pro jejich následné vyhledávání v reálném obraze za pomocí detektorů a deskriptorů klíčových bodů knihovny OpenCV a také dvou dalších nových algoritmů ULATCH[1] a TILDE[2]. Již dříve vznikla na půdě Vysoké školy báňské podobná aplikace tohoto typu, nicméně je napsaná v OpenGL. Impulzem pro tuto práci bylo vydání Vulkan API v roce 2016, jenž má dle svých tvůrců nabízet vyšší výkon a vyváženější využití procesoru či grafické karty oproti starším API typu OpenGL či Direxť3D 11. Vznikla tedy možnost toto API vyzkoušet a zjistit, zda dokáže být pro tento typ úloh vhodnější a rychlejší. V této práci projdu jednotlivé části implementace a popíšu zvolený postup. Celá práce je implementovaná pomocí jazyka C++ a vertex a fragment shadery pomocí jazyka GLSL přeloženého do SPIR-V jazyka, kterému rozumí Vulkan API.

Tato práce bude nejprve pojednávat o samotném Vulkan API a jeho možnostech. Rozeberu jeho silné a slabé stránky a také důvody jeho vzniku. Na několika příkladech rozeberu specifika tohoto API a odlišnosti proti zaběhnutému API typu OpenGL. Také popíšu, proč je aplikace napsána tzv. „headless“. Jedná se o takovou aplikaci, která nemá žádné GUI (Graphical user interface - grafické uživatelské rozhraní). Také přidám svůj osobní pohled na obě API a popíšu rozdíl v implementaci obou API pro část, která se stará o přesun dat z aplikace do OpenCV. Následuje obecný popis realistického osvětlovacího modelu dnes známého jako PBR[3], neboli Physically Based Rendering, který se dnes díky věrnějšímu realistickému ztvárnění zobrazované scény dostává do popředí na poli počítačové grafiky. Po obecném popisu je princip modelu vysvětlen matematicky a následuje také odstavec o využití tohoto modelu v moderních herních enginech typu Unreal Engine 4 nebo Unity, aktuálních možnostech a také předpokládaných přínosech, které by tato technika měla pro mou aplikaci přinést.

V následující části práce se budu zabývat opensource knihovnami, které jsem pro účely této práce využil a nemusel tak psát celou funkcionalitu kompletně od nuly. Jedná se především o jednosouborovou knihovnu TinyObjLoader [4] obstarávající přečtení .obj a .mtl souborů potřebných pro načtení .obj 3D modelů do aplikace. Dále o jednosouborovou knihovnu stbi[5], jenž slouží obecně k manipulaci s obrázky, nicméně pro účely této práce byla použita pouze k vytváření textur čtených ze zdrojových .mtl souborů daných 3D modelů. Jako poslední je využita velmi známá a rozsáhlá open source knihovna OpenCV. Z této knihovny jsem pro mou práci rovněž využil jen zlomek její funcionality. Jedná se například o třídu cv::Mat sloužící pro ukládání hodnot vektorů či matic, které generuje moje aplikace. Také jsem využil několik různých detektorů a deskriptorů klíčových bodů v obraze, které již OpenCV obsahuje. Jedná se zejména o detektory ORB, AKAZE, SIFT, SURF a další.

V další části práce se budu zabývat samotnou implementací a krok po kroku vysvětlím

jednotlivé části implementace a zvolené postupy. Se čtenářem této práce projdu nutné kroky pro inicializaci Vulkan API v aplikaci, nastavení fyzické a logické jednotky, vytvoření command poolu, načtení modelu a zpracování jeho textur, vytvoření vertex a fragment bufferů, vytvoření uniform bufferu, vytvoření grafické pipeline, vytvoření command bufferů a uložení vygenerovaných obrazů z framebufferu do matice OpenCV.

V poslední části krátce představím použité detektory či deskriptory z knihovny OpenCV a také dva novější algoritmy, kterými knihovna OpenCV nedisponuje. Jedná se o algoritmy ULATCH a TILDE. ULATCH je algoritmus pracující jako deskriptor klíčových bodů. TILDE je naopak algoritmus pracující jako detektor klíčových bodů. Tyto algoritmy byly do práce zahrnuty, jelikož jsou založeny na strojovém učení a nejsou starší jak tři roky. Strojové učení je obecně trend, který se v této oblasti dostává stále více ke slovu a naskytla se tak možnost porovnat, zda se tyto algoritmy umístí ve výsledních na předních příčkách. Dále je v této části proveden test, ve kterém se snažíme pozorovat vliv pevně daného šumu v jednom z obrazů na efektivitu algoritmů AKAZE, SURF a TILDE. Tento test je dále rozšířen pro algoritmus AKAZE, kde testuji vliv rozdílné standardní odchylky ve vygenerovaném obrazu se šumem na efektivitu správně vyhledaných klíčových bodů. Výslekdy jsou zaznamenány do grafu. Celá tato část je zakončena tabulkami s porovnáním rychlosti a efektivity jednotlivých algoritmů pro detekci a deskripci klíčových bodů v obrazech. Byla vytvořena také tabulka s nastaveními jednotlivých detektorů či deskriptorů, díky kterým bylo daných výsledků dosaženo.

Výsledkem celé této práce je aplikace, která je schopna pomocí Vulkan API načíst .obj modely s libovolným nastavením pozice kamery, jenž je následně schopna uložit do vyrovnávací paměti aplikace. Z této paměti jsou poté data obrázků kopírována do OpenCV cv::Mat matice a dále mohou být zpracována detektory a deskriptory klíčových bodů. Efektivita a rychlost jednotlivých detektorů je zaznamenána do tabulek a předložena čtenáři. Jelikož je aplikace psána jako tzv. „Proof of concept“, neboli česky „ověření konceptu“, je na závěr také zhodnoceno, zda a jaké přínosy přináší aplikace ve Vulkan API. Na základě této aplikace lze stavět další aplikace podobného typu. Aplikace je zároveň dostupná také jako knihovna a je ji možné připojit k jinému projektu.

2 Vulkan API

V této sekci krátce představím společnost, jenž stojí za Vulkan API. Dále přestavím samotné API a také jeho výhody a nevýhody.

2.1 Khronos Group, Inc.

Khronos Group, Inc. je americké neziskové konsorcium se sídlem v Beavertonu v Oregonu, které se zaměřuje na vytváření otevřených, standardizovaných a bezplatných aplikačních programovacích rozhraní (API) [6]. Společnost byla založena v roce 2000 a v současné době má více jak 100 členských společností [7] mezi něž se řadí například AMD, Apple, Google, Epic Games, Intel nebo Nvidia. Společnost má od roku 2006 ve správě například standardizaci a specifikaci OpenGL API. Dále se společnost zabývá OpenGL|ES, OpenVX, OpenXR nebo NNEF. V roce 2016 Khronos Group, Inc. vydala Vulkan API.

2.2 Co je to Vulkan API?

Vulkan je nové 3D grafické a výpočetní API, které vydala společnost Khronos Group, Inc. v roce 2016. Vulkan API je po dvou letech stále ve vývoji a společnost Khronos Group, Inc. se podpoře tohoto API aktivně věnuje. K datu odevzdání této práce se API nachází již ve verzi 1.1, což je první velká aktualizace API a rozšíření jeho funkcionalit od první verze [8]. Vulkan API cílí hlavně na aplikace vyžadující vysoký výkon v reálném čase. Jedná se například o počítačové hry či interaktivní média nebo složité výpočty. Toto API je multiplatformní a lze s ním vyvíjet jak pro systém Windows, tak Linux nebo Apple MacOS. Pro začátečníky s Vulkan API doporučuji k prostudování knihu Vulkan Cookbook od Pawla Lapinskeho. [9]

2.2.1 Výhody a nevýhody Vulkan API

Vulkan API má své pro a proti. V této krátké podkapitole pár pro a proti vypíšu. Za výhody považuji:

- nízkoúrovňové API
- obrovská možnost ladění a přizpůsobení pro konkrétní účely dané aplikace
- vysoká rychlost a výkon
- validační vrstvy
- nutnost jasné definice velké většiny kódu
- API připravené pro vícevláknové aplikace
- co možná nejmenší režie ovladače grafické karty
- fungování i bez aktivního GUI (více v části headless rendering)

Mezi nevýhody pak patří například:

- velké množství kódu, které je nutné napsat (v porovnání například s OpenGL) než vůbec s pomocí Vulkan API dokážete něco zobrazit na displeji
- oproti vysokoúrovňovým jazykům velmi nekompromisní
- menší podpora komunity vyplývající z mladého věku samotného API

2.3 Validační vrstvy

Vulkan API je navrženo tak, aby představovalo co možná nejmenší režii pro ovladače grafických karet. Díky tomu je ve výchozím nastavení v API velmi omezená kontrola chyb. I jednoduchá chyba (například špatně nastavená hodnota v datovém typu enum) není řešena explicitně a bude mít obvykle za následek pád aplikace nebo její nedefinované chování. Vulkan API vyžaduje, aby byla většina kódu velmi jasně definována, což je na jednu stranu velká výhoda, nicméně na stranu druhou prostor pro mnoho malých chyb. Příkladem může být využití nějaké vlastnosti grafické karty, ale zároveň zapomenutí jejího vyžádání v době, kdy vytváříme ve Vulkan API logické zařízení.

Vulkan proto do svého API zavádí volitelnou součást, tzv. „Validační vrstvy“ (z anglického „Validation layers“). Tyto vrstvy se pak zavěsí na volané funkce API a provádí nad nimi další operace. Důležité je také zmínit, že validační vrstvy jsou přítomny pouze v debug módu aplikace. V release aplikaci se automaticky vypnou a nezatěžují samotnou aplikaci, což přináší další zlepšení výkonu. Častými operacemi jsou například:

- kontrola hodnot jednotlivých parametrů oproti specifikace API pro zjištění špatného využívání
- sledování vytvořených a zničených objektů pro nalezení úniku paměti či jiných zdrojů
- logování každého zavolání funkce a jejich parametrů do standardního výstupu
- trasování volání funkcí pro profilování

Na ukázce níže je vidět příklad validační vrstvy, která kontroluje správnou hodnotu v parametru. V případě, že je jako parametr předán nullpointer, vyskočí chyba.

```
VkResult vkCreateInstance(const VkInstanceCreateInfo* pCreateInfo, const
    VkAllocationCallbacks* pAllocator, VkInstance* instance){
      if (pCreateInfo == nullptr || instance == nullptr)  {
      log("Null pointer passed to required parameter!");
      return VK_ERROR_INITIALIZATION_FAILED;
      }
return real_vkCreateInstance(pCreateInfo, pAllocator, instance);}
```

Výpis 1: Ukázka validační vrstvy ve Vulkan API.

2.4 SPIR-V

V této části se podíváme na další specialitu, kterou Vulkan API má. Jedná se o práci se shadery a formát, ve kterém shadery pracují. Nazývá se SPIR-V a pracuje v bytecode formátu. To je oproti API typu OpenGL velká změna (OpenGL používá GLSL, což je více „user friendly“ formát). Výhodou formátu typu SPIR-V je to, že kompilátory, které programují výrobci grafik pro shadery tohoto formátu, jsou výrazně jednodušší, než ty které jsou programovány například pro OpenGL. Khronos se tímto způsobem snaží zavést unifikovaný standard, pro který budou všichni psát stejný kód shaderů a nebude vznikat tolik problému s různě napsanými shadery každého výrobce a tím také různého chování na různých grafických kartách. Naštěstí nebylo nutné se tento nový formát učit, jelikož Khronos nabízí od základu nativní kompilátor, který není závislý na výrobci žádné grafické karty a který umí převést shader v jazyce GLSL na SPIR-V. Tento kompilátor je navržen tak, aby zkontroloval, že shader napsaný v GLSL jazyce, bude přesně odpovídat jednotnému standardu SPIR-V. Pokud neodpovídá, pak kompilátor vyhodí chybu a nedojde k překompilování jednotlivých shaderů.

2.5 Headless aplikace

Vulkan API od začátku není zamýšleno pouze jako grafické API, ale také výpočetní. Proto jedním z dalších cílů této práce bylo zjistit, zda může aplikace napsaná pomocí Vulkan API běžet kompletně bez aktivního GUI, tedy žádné okno aplikace, ale veškeré výpočty provést na pozadí. Této formě aplikace se říká tzv. „headless“. Vulkan API - oproti stávajícímu API typu OpenGL - tuto možnost skutečně má a celá aplikace v této práci je tímto způsobem napsána. Výhodou možnosti headless aplikace je to, že může být nasazena například na velmi výkonné servery nebo clustery, které obyčejně žádné GUI nemají. Aplikace pak může být ovládána pomocí příkazové řádky, nebo například po síti.

2.6 Srovnání Vulkan API s OpenGL

V této krátké části se pokusím porovnat programování v OpenGL a Vulkan API dle mých zkušeností. Rozdíly jsou patrné už na první pohled a během dvaceti minut, které s oběma API strávíte. OpenGL je vysokoúrovňový jazyk, kdežto Vulkan API je nízkoúrovňový jazyk, což už samo o sobě přináší rozdíly, výhody a nevýhody. Zatímco v OpenGL jste za dvacet minut schopni napsat jednoduchou aplikaci o pár řádcích, která Vám i zobrazí data na displeji počítače, ve Vulkan API s trochou nadsázky pouze nastavíte projekt a vložíte #include knihovnu z Vulkan SDK. V OpenGL je vše připraveno a míra abstrakce je obrovská. Ve Vulkan API je potřeba napsat zhruba 700 řádků kódu, aby bylo možné spustit jednoduchou základní aplikaci. V OpenGL je vše předem nastavené a s některými funkcemi nejde příliš pohnout. Ve Vulkan API musí být vše jasně specifikováno, protože aplikace sama o sobě neudělá nic jiného, než je jí řečeno a nastaveno. Hezký příklad lze uvést na části, ve které se snažím dostat z framebufferu matici, kterou předám do OpenCV k dalšímu zpracování. V OpenGL lze v podstatě využít pouze funkci

glReadPixels s dobře specifikovanými parametry. Tedy všeho všudy jeden řádek. Ve Vulkan API je na stejnou věc potřeba zhruba 110 řádku. Je nutné převést layout obrazu framebufferu na takový, který dovoluje export mimo grafickou kartu, vytvořit vyrovnávací bufferu, do kterého bude obraz nahrán, z vyrovnávacího bufferu provést kopii dat a tyto data následně předat do OpenCV cv::Mat matice. Myslím si, že tohle perfektně vystihuje rozdíl mezi oběma API. Dalším rozdílem je početná komunita a učební zdroje na straně OpenGL versus chybějící zdroje a malá komunita na straně Vulkan API. V tomto pohledu je to nicméně den ode dne lepší s tím jak roste popularita Vulkan API. I přesto, že je Vulkan API na začátku obtížnější se mi s ním pracuje lépe, než s OpenGL a to ze dvou hlavních důvodu. Zaprvé má programátor vše v aplikaci pod svou kontrolou a API samotné mi přijde mnohem srozumitelnější než OpenGL. Zároveň jsou zde i další výhody jako výše zmíněná možnost headless aplikace, multiplatformnost či důraz na výkon a rychlost.

3 Realistický osvětlovací model

Realistický osvětlovací model (z anglického „Physically based rendering“, zkratka PBR) je pokročilá technika stínování a osvětlení v reálném čase. Základem techniky PBR[3] je podrobnější úvaha o chování světla a různých typech povrchů založená více na fyzice než odhadech. Nejprve se podíváme na difúzi s reflexí. Difúze a reflexe jsou v počítačové grafice známy jako základní kámen interakce světla s povrchem. Když světlo narazí na povrch, část z něj se odrazí na protilehlou stranu. Na hladkém povrchu obyčejně dojde k zrcadlovému odrazu. Nicméně ne všechno světlo se od povrchu odrazí a tak se část světelného paprsku může dostat přímo skrz povrch a případně pokračovat ještě dále. Pokud se paprsek světla dostane skrz povrch, pak je možné jej okem dále pozorovat. Tento jev je nazýván difúze. Absorpce a ropztýlení difúzního světla jsou často velmi odlišné na základě vlnové délky daného paprsku, což dává ve výsledku objektům jejich barvu. Pro aproximaci barvy objektu nám zde stačí jedna informace a to difúzní barva (nazývaná také albedo) povrchu.

Následně je nutné vyřešit také průsvitné nebo průhledné objekty, jenž se ve scénách často objevují. Zde už nestačí pouze difúzní barva, ale je potřeba vzít v potaz také tvar či tloušťku nasvíceného objektu. Například při nasvícení tenké vrstvy kůže může dojít k tomu, že světlo projde povrchem kůže a ze zadní strany osvítí další objekty. V tom případě bude kůže lehce průsvitná. V případech velmi nízké difúze povrchu, což je například povrch skla, téměř nedochází k rozptýlení světla a díky tomu se může část scény neporušeně dostat skrze sklo dál. Díky těmto informacím nyní víme, že se difúze a reflexe vzájemně vylučují. Důvod je jednoduchý. Aby mohlo být světlo rozptýleno, musí se nejprve dostat do povrchu, což zase znamená, že se neodrazí od povrchu dál. Tento jev je znám jako úspora energie a zjednodušeně jej lze vysvětlit tak, že světlo které opouští povrch není nikdy jasnější než světlo, které původně na povrch dopadlo. Toho lze v počítačové grafice dosáhnout tak, že před započetím difúzního stínování bude odražené světlo odečteno. To má za následek, že vysoce odrazivé povrchy nebudou vykazovat žádné nebo jen velmi malé rozptylové světlo, což je zapříčiněno tím, že do povrchu proniká jen velmi málo světla a naprostá většina se od povrchu spíše odrazí. A platí to samozřejmě i naopak, tedy pokud má objekt difúzní povrch, pak nebude reflexivní. Tato úspora energie je velmi důležitou součástí PBR stínování a osvětlení.

Dále je nutné vyřešit práci s elektricky vodivými materiály. Jedná se nejčastěji o různé kovy. Kovy mají obecně větší tendenci být mnohem více reflexivnější než nevodiče (například dřevo). Vodiče obvykle vykazují odrazivost okolo 60 až 90 %, zatímco nevodiče spíše v rozmezí 0 až 20 %. Takto vysoká odrazivost bráni tomu, aby se většina světla dostala do okolí a ropztýlila se, což zapříčiní u vodičů velmi lesklý vzhled. Odrazivost u vodičů se také často mění v celém barevném spektru. Jako příklad lze uvést například zlato nebo měď. Při dopadu světla na tyto povrchy je odrazové světlo obarvené dle barvy povrchu. Nevodiče tento jev nevykazují. Jako poslední bod je nutné vyřešit absorbci světla. Vodiče obvykle všechno světlo spíše absorbují než aby došlo k ropztylu světla, které proniká do povrchu. Teoreticky to znamená, že vodiče nebudou nijak

pracovat s difúzním světlem. V praxi to nicméně není pravda, jelikož spousta kovových povrchů je například zoxidovaná nebo jinak porušená a rozptyluje tak malé množství světla. PBR ve svých výpočtech kovovost zahrnuje jako jednu ze složek povrchu. Tvůrci modelů či textur by měli uvádět, jak moc se u daného modelu či textury jedná o kovový povrch.

Jako další je nutné vyřešit Fresnelovy rovnice, které udávají intenzitu odraženého a lomeného světla. Fresnelovy rovnice jsou v počítačové grafice obecně známy již dlouhé roky, nicméně technika PBR chování Fresnelových rovnic mírně upravuje. První změnou je, že každý objekt, který má hladký povrch (a tedy dobrou odrazivost), může po svých krajích či hranách sloužit jako zrcadlo bez ohladu na to, o jaký se jedná materiál. Nutný je pouze dobrý úhel pohledu a hladká úprava povrchu. Další změnou je pozorování, že mezi různými materiály není v rámci křivky mezi úhly Fresnelových rovnic výrazný rozdíl. To vede ke dvěma poznáním. Jednak v rámci PBR víme, kde se zhruba nachází výchozí hodnoty Fresnelovy rovnice pro materiály a také víme, že pokud je cílem dosáhnout realismu, je potřeba omezit tvůrcům modelů či textur přístup k úpravě této hodnoty. U každého materiálu je potřeba definovat základní odrazivost, což poskytne minimální množství odraženého světla a jeho barvu. Fresnelovy rovnice přidají k již definované odrazivosti větší odrazivost a díky tomu jsou některé povrchy více reflexivní pod různými úhly.

Jednou s posledních částí techniky PBR je řešení mikročástic. Výše uvedený popis difúze a odrazu světla obvykle záleží na orientaci povrchu, na který světlo dopadá. Často je to suplováno tvarem načtených modelů, které navíc obsahují normálové textury. Toto jsou pro většinu klasických renderovacích technik dostačující informace, se kterými může renderování difúze a reflexe proběhnout. U PBR to bohužel nestačí. V reálném světě často dochází k tomu, že povrchy těles mají velmi malé nedostatky. Jedná se o drobné trhlinky, rýhy, hrudky a jiné. Často jsou tak malé, že je lidské oko ani nepostřehne a není možné je zobrazit v normálové mapě s rozumným rozlišením. I přesto, že tyto nedostatky nejsou vidět, ovlivňují putování světla scénou. Nejvíce zřetelný vliv mají tyto mikročástice na reflexi. Velmi jednoduchým příkladem to lze opět přirovnat k míči. Pokud hodíme míč například na skálu s nerovnoměrným povrchem, tak se míč vždy odrazí, ale nikdy nevíme kam přesně to bude. A stejné je to s povrchem objektů. Čím hrubší daný povrch je, tím více se odrážené světlo bude lišit nebo bude rozmazané. Bohužel dnešní technika stále ještě není na takové úrovni, abychom mohli vypočítat každý paprsek světla s každou mikročásticí, a proto technika PBR specifikuje obecnou hodnotu udávající hrubost povrchu, se kterou mohou shadery pracovat. Tato hodnota je specifikována buď jako textura nebo jako konstanta.

Vzhledem k tomu, že nyní počítáme s hrubostí povrchu, je nutné zajistit, aby se od povrchu odráželo správné množství světla. Tato část je pro PBR velmi důležitá. PBR se snaží o to, aby u povrchů, které jsou hrubé, docházelo k rozsáhlejšímu, méně ostrému odrazu světla. Naopak u méně hrubých materiálů bude docházet k menšímu rozsahu rozptylu světla, ale o to bude ostřejší. Oba typy povrchy tedy odráží stejné množství světla, ale každý z nich jiným způsobem. Hrubší povrch světlo rozprostře na větší plochu v několika směrech, bude tedy působit spíš

rozmazaně, nebo neurčitě. Hladký povrch bude naopak odrážet spíše koncentrovaný paprsek a bude tak ostřeji vidět, co odráží. Tento bod je v zásadě druhá forma úspory energie. Díky tomuto poslednímu bodu si uvědomíme, že lesk mikročástic v objektech scény ovlivňuje jas reflexe. To je velmi důležité z pohledu reálného fungování stínování a osvětlení. Technika PBR bere odrazivost a detaily mikročástic jako dvě spojené veličiny, se kterými je nutné pracovat již při výrobě modelů či textur a také při samotném stínování a osvětlení.

Matematický výpočet realistického osvětlovací modelu

Jak již bylo řečeno, v PBR se vše točí okolo mikročástic. Žádný povrch není nikdy kompletně hladký (pokud mluvíme právě o mikroskopických detailech). V PBR pro určení hrubosti povrchu slouží parametr roughness. Na základě tohoto čísla můžeme zhruba vypočítat poměř mikročástic zarovnaných s vektorem h. Tento vektor h je nazýván poloviční vektor a nachází se přesně v polovině mezi pozorovacím úhlem v a vyzářeným úhlem světla l. Výpočet poločního vektoru provedeme pomocí následující rovnice

$$h = \frac{l + v}{\|l + v\|} \ . \tag{1}$$

Dále se dostáváme ke komplikované rovnici, jenž je známa jako „zobrazovací rovnice“ (z anglického Rendering equation). Zobrazovací rovnice je aktuálně nejlepší model, dle kterého se snažíme simulovat přenos světla ve scéně. PBR velmi silně vychází z více specializované verze této rovnice. Nicméně, abychom pochopili PBR, je potřeba se nejprve podívat na rovnici odrazu

$$L_o(p, \omega_o) = \int_\Omega f_r(p, \omega_i, \omega_o) L_i(p, \omega_i) n \cdot \omega_i d\omega_i \ . \tag{2}$$

Rovnice odrazu se z prvu zdá jako velmi složitá, nicméně po rozklíčování všech části nám bude vše dávat smysl. K pochopení rovnice je potřeba na chvíli odbočit a podívat se na radiometrii. Radiometrie je měření elektromagnetického záření včetně viditelného světla. Existuje několik radiometrických veličin, které můžeme k měření světla používat, ale pro PBR je zde důležitá pouze jedna a to síla vyzařovaného světla (nebo také jas). Tato veličina je ve výše uvedené rovnici označena jako L. Síla vyzařovaného světla se používá pro kvantifikaci velikosti nebo síly světla, které vychází z nějakého směru. Záření samotné je kombinací více fyzických veličin.

Nejprve je nutné se podívat na zářivý tok. Zářivý tok je přenášená energie z nějakého světelného zdroje. Tato energie je měřena ve wattech. Světlo je souhrný součet energie na více různých vlnových délkách a každá vlnová délka je spojená s určitou viditelnou barvou. Na vyzařovanou energii z určitého zdroje světla lze pak nahlížet jako na funkci všech obsažených přípustných vlnových délek. Vlnové délky mezi 390 až 700 nanometry jsou považovány za součást viditelného barevného spektra. Jedná se o takové vlnové délky, které dokáže postřehnout lidské oko. Zářivý tok tedy vyjadřuje absolutní energetický obsah záření. Jelikož je samotné měření záření pro počítačovou grafiku mírně nepraktické, používáme místo toho jako hodnotu jednoduše barvu

světla ve formátu RGB. Dále je potřeba vyřešit tzv. prostorový úhel. Tento úhel je v rovnici zastoupen jako w a reprezentuje velikost nebo spíše oblast objektu, který promítáme na scénu. Následně je potřeba vyřešit intenzitu záření. Intenzita záření je v podstatě informace o tom, jak silné je světlo vyzařované do scény. Intenzita světla je vypočtena rovnicí

$$I = \frac{d\phi}{d\omega}\,, \tag{3}$$

kde hodnota ϕ reprezentuje zářivý tok a hodnota ω prostorový úhel. Se znalostí těchto tří složek je nyní možné vypočítat rovnici záření

$$L = \frac{d^2\phi}{dAd\omega\cos\theta}\,. \tag{4}$$

Tato rovnice je popsána jako celková pozorovaná energie v oblasti A s prostorovým úhlem ω s intenzitou světla ϕ. Záření je radiometrická veličina definovaná jako množství světla v oblasti měřené úhlem světla θ k normále povrchu $\cos(\theta)$. Čím slabší je světlo, tím menší je také jeho vyzařování a naopak. Rovnice záření je zde velmi důležitá, neboť se skládá z většiny fyzikálních veličin, které pro PBR potřebujeme. Pokud budeme považovat prostorový úhel ω a oblast A za nekonečně malou, můžeme využít záření pro změření zářivého toku jednoho paprsku světla protínajícího jeden bod v prostoru. Tento vztah nám dovolí vypočítat intenzitu záření jednoho paprsku světla, který bude ovlivňovat jeden bod (fragment). Prostorový úhel ω tak můžeme přeložit na směrový vektor ω a oblast A na bod p. Díky tomu můžeme sílu záření použít přímo v shaderech aplikace k výpočtu príspěvku síly jednoho paprsku světla na jeden fragment. V řeči shaderů si lze představit výpočet této funkce následujícím způsobem.

```
int steps = 100;
float sum = 0.0f;
vec3 P   = CalculateVectorP();
vec3 Wo  = CalculateVectorWo();
vec3 N   = CalculateVectorN();
float dW = 1.0f / steps;

for(int i = 0; i < steps; ++i)
   {
      vec3 Wi = getNextIncomingLightDir(i);
      sum += Fr(P, Wi, Wo) * L(P, Wi) * dot(N, Wi) * dW;
   }
```

Výpis 2: Ukázka příkladu shaderu pro výpočet intenzity záření.

Zbývá nám vypočítat hodnotu f_r, jenž je symbolem pro obousměrnou distribuční funkci odrazu světla (z anglického Bidirectional Reflectance Distribution Function, zkratka BRDF). Tato funkce se používá v počítačové grafice jako matematické vyjádření vlastností povrchu.

Udává hustotu pravděpodobnosti, že se světlo, které na povrch dopadne, odrazí daným směrem. Mezi parametry této funkce patří příchozí směr ω_i a odchozí směr ω_o. Oba tyto směry jsou definované vůči dalšímu parametru, jímž je normála povrchu n. Funkce také obsahuje parametr α, jenž reprezentuje hodnotu hrubosti mikročástic povrchu. BRDF se snaží odhadnout poměr odraženého záření vůči ozáření povrchu. Aby byla tato rovnice fyzicky věrná, musí respektovat zákon úspory energie zmiňovaný výše. Ve zkratce jde o to, že součet odraženého světla by nikdy neměl překročit množství příchozího světla. Aktuálně je možné vybírat z více druhů BRDF rovnic, nicméně většina real time aplikací využívá Cook-Torrance BRDF[12]. Jejich rovnice

$$f_r = k_d f_{lambert} + k_s f_{cook-torrance} \ , \tag{5}$$

obsahuje jak difúzní, tak speculární složku. Zde je k_d dříve zmíněný poměr příchozího světla ku světlu, které se zalomí a k_s je poměr příchozího světla k odraženému světlu. Levá strana této rovnice vyjadřuje difúzní složku, která je označena jako $f_{lambert}$. Jedná se o tzv. Lambertův zákon

$$f_{lambert} = \frac{c}{\pi} \ . \tag{6}$$

V této rovnici je c albedo barvou nebo taky barvou povrchu. Tato hodnota je vydělena hodnotou π, aby došlo k normalizaci difúzního světla. Tato normalizace je nutná, jelikož π je využíváno pro škálování integrálu, jenž obsahuje BRDF funkci. Samozřejmě existuje několik různých rovnic, které řeší difúzní složku BRDF, která může vypadat i více realisticky, nicméně je i více výpočetně náročná. Jelikož je Lambertův zákon pro většinu real time aplikací plně dostačující, což potvrdil i Brian Karis z Epic Games ve svém výzkumu nasazení PBR do Unreal Engine 4, není potřeba hledat jinou alternativu.[3] Speculární složka je mírně složitější rovnicí

$$f_{cook-torrance} = \frac{DFG}{4(\omega_o \cdot n)(\omega_i \cdot n)} \ . \tag{7}$$

Tato speculární část obsahuje celkem tři funkce. Každý ze symbolů D, F, G reprezentuje typ funkce, která odhaduje specifickou část reflektivních vlastnostní povrchu. Jedná se o funkci normální distribuce (rozdělení) D, Fresnelovu rovnici F a geometrickou funkci G. Normální funkce D odhaduje množství mikročástic, které jsou zarovnány s poloviční vektorem ovlivněným hrubostí povrchu. Fresnelova rovnice F popisuje poměr reflekce povrchu k různým úhlům povrchu. Geometrická funkce G popisuje vlastnost samozastínění mikročástic. Pokud je povrch objektu dost hrubý, může dojít k tomu, že se jednotlivé mikročástice překrývají a díky tomu může dojít k redukce množství odraženého světla. Jako první se podíváme na výpočet normální distribuční funkce

$$NDF_{GGXTR}(n, h, a) = \frac{\alpha^2}{\pi((n \cdot h)^2(\alpha^2 - 1) + 1)^2} \ , \tag{8}$$

kde je h opět poloviční vektor a α je hodnota hrubosti povrchu. Když je hrubost povrchu malá (tedy povrch je téměř hladký), pak je velké množství mikročástic zarovnáno v malém rádiusu kolem poloviního vektoru a silně září. Naopak, když je hrubost velká, mikročástice se k poloviničnímu vektoru tolik nezarovnávají a vyzařované světlo je spíše rozprostřeno do okolí. Normální distribuční funkce D může v shaderech vypadat například takto.

```
float DistributionGGX(vec3 N, vec3 H, float a)
{
   float a2    = a*a;
   float NdotH = max(dot(N, H), 0.0);
   float NdotH2 = NdotH*NdotH;

   float nom   = a2;
   float denom = (NdotH2 * (a2 - 1.0) + 1.0);
   denom       = PI * denom * denom;

   return nom / denom;
}
```

Výpis 3: Ukázka normální distribuční funkce D pro PBR.

Nyní je potřeba vypočítat geometrickou funkci G. Tato funkce stejně jako předchozí funkce D bere jako vstupní parametr hrubost povrchu. Geometrická funkce v PBR je obecně používána v kombinaci s GGX a Schlick-Beckmanovou aproximací známou jako Schlick-GGX

$$G_{SchlickGGX}(n, v, k) = \frac{n \cdot v}{(n \cdot v)(1 - k) + k} \,. \tag{9}$$

V této rovnici je k přemapovaná α, tedy hodnota hrubosti povrchu založená na tom, zda používáme přímé osvětlení, nebo osvětlení na základě IBL[13] (Image Based Lighting). Abychom dokázali geometrii efektivně aproximovat, musíme vzít v potaz jak směrový vektor, tak vektor odrazu světla a vypočítat rovnici

$$G(n, v, l, k) = G_{sub}(n, v, k)G_{sub}(n, l, k) \,. \tag{10}$$

Jako poslední zbývá vyřešit Fresnelovu rovnici. Tato rovnice popisuje poměr mezi lomeným a odraženým světlem, což se liší dle úhlu, pod kterým světlo na povrch dopadá. Pro výpočet této rovnice je využita Fresnel-Schlick aproximace

$$F_{Schlick}(n, v, F_0) = F_0 + (1 - F_0)(1 - (n \cdot v))^5 \,, \tag{11}$$

kde F_0 reprezentuje základní odrazivost povrchu, což je vypočteno pomocí indexu lomu. Všechny ostatní parametry již byly vysvětleny a nezbývá než ukázat finální rovnici pro výpočet PBR osvětlení založenou na Cook-Torrancově BRDF

$$L_o(p, \omega_o) = \int_\Omega (k_d \frac{c}{\pi} + k_s \frac{DFG}{4(\omega_o \cdot n)(\omega_i \cdot n)}) L_i(p, \omega_i) n \cdot \omega_i d\omega_i \ . \tag{12}$$

Tato rovnice nyní kompletně popisuje výpočet PBR, neboli realistického osvětlovacího modelu, který je v poslední době na poli počítačové grafiky hojně využíván. Tento výpočet byl realizovaný například v herním enginu Unreal Engine 4.

Využití PBR v praxi

V roce 2018 je PBR hojně zmiňovaným pojmem především v oblasti počítavé grafiky a výroby počítavých her. Všechny větší herní enginy jako Unity, Unreal Engine 4 nebo CryEngine 5 již podporu pro PBR obsahují a je tak velmi jednoduché začít s PBR pracovat, aniž by bylo potřeba důkladně znát samotné fungování tohoto stínování a osvětlení. Je pouze na jednotlivci či týmu, zda těchto možností využije. Největší nevýhoda PBR techniky je totiž právě v její složité implementaci. V případě, že je cílem aplikace či počítačová hra, která s PBR pracuje, je nutné tomu od začátku přizpůsobit proces, jelikož i objekty (3D modely, textury atd.) musí být pro tento typ stínování a osvětlení upraveny jinak, než bylo doposud zvykem. Paradoxně je výroba nových modelů či textur pro PBR jednodušší, jelikož dochází k výrobě pouze jedné textury, ve které jsou uvedeny všechny informace. U klasických materiálů je obecně potřeba připravit zvlášť difúzní, speculární či normálovou mapu. PBR se těší stále větší popularitě, což je dle mého názoru zapříčiněno hlavně tím, že byla odbourána implementační bariéra právě díky herním enginům, které již podporu PBR obsahují. Tyto herní enginy stačí v dnešní době nainstalovat a začít k nim vytvářet PBR kompatibilní zdroje.

Výhody implementace PBR

Základní implementace mé aplikace pracuje s Phongovým osvětlovacím modelem[10]. Tento starší osvětlovací model je velmi populární a používaný jako základ ve spoustě dnešních aplikací. Dá se říct, že dnes je již překonaný. Existují i jiné typy osvětlovacích modelů, například Blinn-Phongův osvětlovací model[11]. Oba tyto osvětlovací modely nicméně spojuje to, že nejsou založeny na reálných fyzických vlastnostech. Z dvou výše uvedených kapitol je patrné, čeho se obecně v aplikacích snažíme nasazením PBR osvětlovací modelu docílit. Jedná se hlavně o reálnější a věrnější podobu vygenerovaného modelu či dobrého prosvícení scény. Konkrétně u aplikace v mé diplomové práci by mělo být očekávaným efektem implementace PBR taktéž zkvalitnění vygenerovaného modelu, což by ve výsledku mělo přinést větší počet správně spárovaných klíčových bodů a tedy větší efektivitu všech měřených algoritmů. Větší efektivita by se mohla projevit také v případě, kdy je načtenému modelu dynamicky měněna pozice kamery a generovaný obraz spolu s reálnou fotografií tak nemají stejnou velikost, pozici, či rotaci modelu.

4 Použité knihovny

Třetí část mé práce se zabývá krátkým popisem knihoven, které jsem využil. Některé z nich jsem využil méně, některé více. Dané knihovny jsem použil z důvodu usnadnění a zrychlení práce. U každé knihovny popíšu, jak jsem ji využil a v čem byla přínosem.

Tinyobjloader

Vývoj této knihovny začal v roce 2012 jejím autorem Syoyo Fujitou. Jedná se o knihovnu, která je napsaná v jazyce C++ v jediném malém souboru. Nemá žádné speciální závislosti na jiných souborech či knihovnách. Jediná závislost, kterou ke svému fungování potřebuje, je knihovna C++ STL (standard template library). Knihovna je volně ke stažení na adrese `https://github.com/syoyo/tinyobjloader` a díky tomu, že je hostována na Githubu do ní již také přišpělo velké množství dalších přispěvovatelů. V srpnu roku 2016 vyšla nová verze knihovny, která přinesla novou datovou strukturu a také vylepšené API. Tinyobjloader dokáže sparsovat až 10 milionů polygonů s průměrnou spotřebou paměti a času potřebného ke zpracování. Knihovna dokáže zpracovat tyto částí .obj a .mtl souborů:

- skupiny modelu
- vertexy
- souřadnice textur
- normály
- materiály
- PBR (Physically based rendering) rozšíření pro materiály
- smoothing groups

Jako další vlastnosti této knihovny lze vypsat vlastní callback API pro načítání modelů nebo „double precision support“ pro HPC (High Performance Computing).

Poslední zajímavostí tinyobjloaderu je experimentální sestavení, ve kterém je možné načítat 3D .obj modely pomocí více vláken. Autor knihovny u tohoto experimentálního sestavení uvádí, že je zhruba 4,5x rychlejší, než základní jednovláknová knihovna. [4]

Stbi-image

Jedná se o malou standalone knihovnu pocházející z větší skupiny knihoven stb `https://github.com/nothings/stb`, která pracuje s obrazovými formáty různého typu. Jedná se především o JPG, PNG, TGA, BMP, PSD, GIF, HDR, PIC. Knihovna se stará převážně o načtení či dekódování obrazových dat z disku či paměti počítače. Autorem této knihovny je Sean T. Barrett. Z jeho jména také vychází jméno pro všechny další knihovny pod hlavičkou stb. Pro zajímavost uvádím pár dalších knihoven a jejich možnosti.

- stb_vorbis - dekódování ogg vorbis souborů z disku či paměti počítače na float hodnoty
- stb_image_write - zapisování obrázků typu PNG, TGA, BMP na disk počítače
- stb_voxel_render - voxelový renderovací engine na styl počítačové hry Minecraft
- stb_truetype - parsování, dekódování a rasterizování znaků z truetype fontů

GLM

GLM, celým názvem OpenGL Mathematics je podpůrná matematická knihovna, která byla navržena pro práci s OpenGL a je založená na specifikaci GLSL[14]. GLM poskytuje třídy a funkce navržené a implementované se stejným stylem pojmenování jako GLSL. Díky tomu je pro každého, kdo již zná GLSL velmi jednoduché začít používat knihovnu GLM. Knihovna samotná nicméně není navržena striktně pro funkce GLSL. GLM obsahuje systém rozšíření, který poskytuje další rozšířené funkce. Jedná se například o transformace matic, kvaterniony, generování náhodných čísel, generování šumu atd. V mé práci tuto knihovnu používáme hlavně pro MVP matici.

OpenCV

OpenCV neboli Open Source Computer Vision Library je velká volně šířená softwarová knihovna zabývající se převážně technikami rozpoznávání obrazu a strojovým učením. Cílem této knihovny je od začátku poskytnout jakýsi základní můstek či infrastrukturu pro aplikace využívající rozpoznávání obrazu nebo strojové učení i s ohledem na využití v komerční sféře.

Knihovna obsahuje více jak 2500 různých algoritmů [15]. Tyto algoritmy mohou být využity například pro detekci tváří, identifikaci objektů, klasifikaci činností na videu, sledování pohybu kamery, sledování hýbajících se objektů, extrahování 3D objektů ze scény, tvoření mraků 3D bodů ze stereo kamer, spojení více obrázků do jednoho pro vytvoření jednoho obrázku ve vysokém rozlišení, hledání podobných obrázků v databázích, odstranění efektu červených očí způsobeného bleskem fotoaparátu, sledování pohybu očí, rozpoznávání scény a možnosti obohacení scény pomocí rozšířené reality a jiné. OpenCV komunita se skládá z více jak 47 tisíc členů a odhadovaný počet stažení této knihovny je kolem 14 milionů [15]. Tato knihovna je značně využívána ve firmách, výzkumných skupinách i vládních orgánech. Jako příklad lze uvést například firmy typu Google, Yahoo, Microsoft, Intel, IBM, Sony, Honda, Toyota. OpenCV stojí například za spojováním jednotlivých obrázků v Google StreetView, monitorováním důlního nářadí v Číně, detekcí možných případů utonutí v bazénech v Evropě nebo například kontrolu čistoty přistávacích drah letišť v Turecku.

OpenCV má nativně rozhraní pro C++, Python, Javu a MATLAB. Podporuje systémy Windows, MacOS, Linux a Android. Samotná knihovna je napsána v jazyce C++.

5 Implemetace generátoru syntetických 3D modelů pomocí Vulkan API

V této části podrobně projdu vytvoření aplikace pro generování syntetických 3D modelů pomocí Vulkan API. K tomu, aby aplikace dokázala takový model vygenerovat a zobrazit, vede několik kroků.

5.1 Vytvoření instance, fyzické a logické jednotky, command pool

Nejprve je potřeba vytvořit Vulkan instanci. Následně fyzickou a logickou jednotku, se kterou bude aplikace pracovat. Jako poslední v této části vytvoříme command pool.

5.1.1 Vytvoření instance

Vytvoření instance je první částí k vytvoření aplikace pomocí Vulkan API. Vytvořením instance dochází zároveň k inicializaci Vulkan knihovny. V této části je možné specifikovat volitelnou strukturu VkApplicationInfo. Zde je možné vyplnit například název aplikace nebo jméno použitého enginu, což se hodí v případech, kdy chceme ovladači grafiky říci, že používáme nějaký etablovaný engine jako například CryEngine nebo Unreal Engine. Tyto enginy často obsahují speciální funkce či chování, pro které může ovladač naši aplikaci rovnou optimalizovat. Dále je mimo jiné možné specifikovat minimální verzi Vulkan API, pro kterou je aplikace psána. Pokud není verze API vyplněna, pak lze teoreticky aplikaci spustit na jakékoliv verzi API, nicméně může dojít k nedefinovanému chování nebo pádům aplikace například z důvodů chybějící funkcionality vyšších verzí.

Nutnou součástí Vulkan instance je vytvoření struktury VkInstanceCreateInfo. Zde je explicitně nastaveno, jaký typ instance bude použit.
Jedná se například o VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO) a jsou zde předány všechny vyplněné data z předchozí struktury VkApplicationInfo.

Vulkan API je samo o sobě nezávislé na platformě. Pokud je cílem aplikace například zobrazovat vyrenderovaný obsah framebufferu, je potřeba sáhnout po rozšíření, které tuto práci obstará. Pro systém Windows se můžeme bavit například o knihovně GLFW.

5.1.2 Volba fyzické jednotky (GPU)

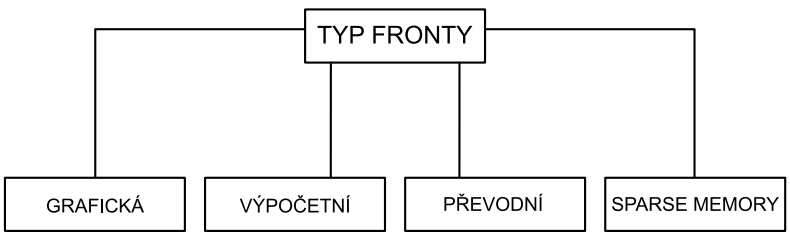
Další krokem je volba fyzického zařízení, tedy grafické karty. Vulkan API vždy nejprve pomocí funkce vkEnumeratePhysicalDevices zjistí, zda a jaké karty jsou v daném přístroji dostupné a jaké podporují rozšíření. V případě mé aplikace pak vždy vyberu první nalezenou grafickou kartu, jelikož počítám s tím, že aplikace bude testována na počítači s jednou kartou. V této části je také místo pro zjišťování podpory vlastností, které má daná grafická karta.

5.1.3 Volba logické jednotky

Poté, co je vybrána fyzická jednotka, je potřeba vybrat také logickou jednotku. Logická jednotka slouží jako rozhraní pro fyzickou jednotku (grafickou kartu). Je důležité poznamenat, že v případě nutnosti (například rozdílné požadavky aplikace) je možné k jedné fyzické jednotce vytvořit více logických jednotek.

Pro zdárné vytvoření logické jednotky je stejně jako v případě fyzické jednotky nutné vytvořit a naplnit daty konkrétní strukturu. Zde se jedná hlavně o strukturu VkDeviceQueueCreateInfo, ve které je zapotřebí explicitně specifikovat jakou budeme používat „skupinu front“ (z anglického „queue family“) a počet těchto front. Skupiny front spolu sdílí stejné nastavení a možnosti pro zpracování příkazů. Fronty jsou v podstatě komunikačním prostředníkem mezi grafickou kartou a aplikací. Vulkan pracuje celkem se čtyřmi frontami a každá skupina front umí pracovat s jiným typem příkazů, které přijímá pomocí command bufferů, o kterých se více rozepisuji v další části této práce. Každá grafická karta může obsahovat různé skupiny front a také různé počty samotných front uvnitř těchto skupin. Například novější grafické karty Nvidia obyčejně obsahují dvě různé skupiny front – „hybridní“ a transportní. Pojem hybridní používám, protože tato skupina dokáže zpracovávat všechny typy příkazů. Hybridní skupina má v případe karet Nvidia celkem šestnáct samostatných front a transportní skupina obsahuje pouze jednu transportní frontu používanou pro transfer dat. Naproti tomu novější grafické karty výrobce AMD jsou většinou rozděleny na tři fronty, kde taktéž existuje jedna hybridní skupina, ale pouze s jednou frontou. K této skupině se připojuje druhá skupina, ve které lze řešit výpočetní a transférové příkazy v počtu dvou front a třetí je skupina pro transfery, taktéž v počtu dvou front.

Před vytvořením samotné logické jednotky dojde ke zjištění, jaké skupiny front je schopna grafická karta používat. Tyto informace jsou poté vloženy do struktury VkDeviceCreateInfo. Pokud jsme se při tvorbě fyzické jednotky dotazovali na speciální vlastnosti grafické karty (například pokud chceme použít geometrické shadery), pak na tomto místě musíme dané vlastnosti do struktury VKDeviceCreateInfo vložit jako seznam vlastností k použití. V případě aplikace v této diplomové práci se spokojíme pouze s jednou frontou grafického typu. Vlastnosti jednotlivých grafických karet lze zjišťovat například na webové stránce `http://vulkan.gpuinfo.org/listreports.php`



Obrázek 1: Typy využitelných front ve Vulkan API.

5.1.4 Command pool

Command pool je možné popsat jako manažera paměti, který využívají buffery. Z toho vyplývá, že command pool musí být vždy vytvořen dříve, než command buffery. Ty navíc alokují svou pamsěť právě z command poolu.

Jak je již u Vulkan API zvykem, i zde je potřeba vytvořit command pool pomocí jisté struktury – VkCommandPoolCreateInfo. Nicméně dál není potřeba vytvářet nic speciálního a lze tak rovnou přistoupit k vytvoření command poolu pomocí funkce vkCreateCommandPool.

5.2 Načtení, zpracování a příprava 3D modelu včetně zpracování textur

Tato část se zabývá načtením, zpracováním a připravením 3D modelu ve formátu .obj pro moji aplikaci. Podrobně také popíšu vytváření textur pro model.

5.2.1 Načtení 3D modelu

V této části se budeme zabývat hned několika body. Jedná se především o načtení dat samotného modelu, vytvoření potřebných textur pro model a nakonec integraci dat do mé aplikace.

Po vytvoření command poolu se dostáváme k rozsáhlejší části, kde dochází k načtení 3D modelu ve formátu .obj včetně jeho materiálového souboru .mtl i s přiloženými texturami. Pro načtení 3D modelu jsem použil knihovnu tinyobjloader, jež je popsána v samostatné kapitole (ODKAZ na kapitolu od tinyobjloaderu). Tinyobjloader operuje s vlastními datovými strukturami, jako jsou:

- tinyobj::attrib_t
- tinyobj::shape_t
- tinyobj::material_t

Struktura attrib_t obsahuje informace o vrcholech, normálách, souřadnících textur a barvě. Jednotlivé informace jsou udržovány ve vlastních vekrotech attrib.vertices, attrib.normals a attrib.texcoords. Struktura shape_t obsahuje informace o jednotlivých částech modelu a jeho plochách. Každá taková plocha se skládá z pole vrcholů a každý vrchol obsahuje indexy na jeho pozici, normály a souřadnice textur. Nakonec zbývá struktura material_t, která, jak už název napovídá, obsahuje informace o materiálech a je co se týče velikosti největší. Tato struktura obsahuje například název souboru konkrétní textury na disku nebo jednotlivé složky, ze kterých se daný materiál skládá (ambient, specular, diffuse, transmittance, emission, shininess, ior, dissolve, illum, cesty k diffuzním, speculární či dalším texturám atd.). Tinyobjloader je připravený i na parametry pro tzv. PBR (Physicaly Based Rendering) renderování. Jako další je potřeba zmínit string::err, jenž je využit jako čtvrtý parametr a jeho účelem je vypsání chyb nebo upozornění, které vznikly při načítání modelu knihovnou tinyobjloader. Nakonec zbývají dva poslední parametry, které jsou pro načtení modelů důležité. Jedná se o cestu k modelu, tedy samotný .obj

soubor a následně cestu k materiálům, tedy .mtl soubor daného 3D modelu. Tyto dvě proměnné jsou definovány globálně.

Načtení samotného modelu do paměti probíhá následující funkcí:

```
if (!tinyobj::LoadObj(&attribs, &shapes, &mats, &err, MODEL_PATH.c_str(),
    MATERIAL_PATH.c_str())) {
throw std::runtime_error(err);
}
```

Výpis 4: Ukázka načtení 3D modelu do pamětí pomoci knihovny tinyobjloader.

Po úspěšném načtení modelu nejprve zpracuji všechny materiály, které model obsahuje. Pro tyto účely jsem využil vlastní datové struktury Material, která obsahuje tyto parametry:

- static std::unordered_map<uint64_t, Material> materials
- std::string name
- glm::vec3 ambient
- glm::vec3 diffuse
- glm::vec3 emissive
- glm::vec3 specular
- float reflection
- float opacity
- float shininess

Pro tuto strukturu jsem následně vytvořil vector<Material> materials, který plním daty z právě načteného modelu. U všech materiálu dojde ke zpracování parametrů name, ambient, diffuse, emissive, specular, opacity a reflection.

5.2.2 Zpracování textury modelu

V rámci zpracování všech materiálů modelu se dostáváme také k materiálům, které obsahují difůzní texturu, tedy parametr „map_kd“. U takových materiálů rovnou v paměti tuto texturu vytvářím a ukládám ji do vektoru vector<ModelTextures> modelTextures. Pro textury jsem vytvořil datovou strukturu ModelTexture, která obsahuje tyto parametry:

- unsigned int id
- std::string name
- std::string path
- VkImage textureImage

- VkDeviceMemory textureImageMemory
- VkImageView textureImageView
- VkSampler textureSampler
- void CreateTextureImage(const char* texturePath)
- void CreateTextureImageView()
- void CreateTextureSampler()

Vytvoření textury je oproti OpenGL výrazně složitější. Vulkan API je opět potřeba explicitně říct jakým způsobem se budou vytvářet obrázky a jaké budou mít parametry, informovat o vytvoření bufferu či samotných obrázků, vytvořit command buffery, změnit uspořádání (z anglického „layout“) obrázku a nakonec založit tzv. „ImageView“ a také „ImageSampler“. Na následujících řádcích proces vytvoření popíšu.

5.2.3 Vytvoření textury

Při nalezení materiálu, který obsahuje difůzní složku (parametr „map_Kd) ihned v paměti dochází k založení obrázku s danou texturou. Nejprve je volána funkce void CreateTextureImage(cont char* texturePath) Tato funkce ve svém parametru předává adresu nalezené textury na disku. Předtím, než začne proces vytvoření samotného obrázku s texturou je potřeba definovat proměnné výška, šířka a počet kanálů, do kterých budou uloženy informace z načtené textury. Nyní přichází na řadu jediné místo, kde je v celém projektu využita knihovna stb_image, která je popsána v samostatné části [4] výše. Pomocí následující funkce dojde k načtení souboru z disku do paměti.

```
stbi_uc* pixels = stbi_load(texturePath, &texWidth, &texHeight, &texChannels,
    STBI_rgb_alpha);
```

Výpis 5: Ukázka načtení jedné textury 3D modelu.

Funkce stbi_load obsahuje tyto parametry:

- texturePath - cesta k textuře na disku
- texWidth – šířka textury
- texHeight – výška textury
- textChannels – počet kanálů, které textura obsahuje
- STBI_rgb_alpha – požadovaný formát obrázku, se kterým budeme dále pracovat (stbi knihovna tento formát vynutí, i když textura žádný alpha kanál nemá, což se hodí pro zachování konzistence formátu všech textur)

Po načtení textury je potřeba explicitně alokovat množství paměti, které bude potřeba pro Vulkan. To lze provést pomocí následujícího kódu.

```
VkDeviceSize imageSize = texWidth * texHeight * 4;
```

Výpis 6: Ukázka alokace paměti pro texturu.

Nyní je potřeba založit vyrovnávací buffer (z anglického „staging buffer“). Lze využít i VkImage (obraz), nicméně buffer je doporučené řešení a také je na některých hardwarech rychlejší. [16]. Tento buffer je založen následující pomocnou funkcí.

```
createBuffer(VK_BUFFER_USAGE_TRANSFER_SRC_BIT,
   VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT|VK_MEMORY_PROPERTY_HOST_COHERENT_BIT,
         &stagingBuffer,
         &stagingBufferMemory,
         imageSize);
```

Výpis 7: Ukázka vytvoření vyrovnávacího bufferu pro vytvoření textury.

Potřebné funkce pro vytvoření bufferu jsou:

- usageFlags – jaký bude typ použití
- memoryPropertyFlags – jakým způsobem bude řešena paměť bufferu
- VkBuffer – pointer na objekt vyrovnávacího bufferu
- VkDeviceMemory – pointer na paměť vyrovnávacího bufferu
- VkDeviceSize – velikost v bytech pro alokaci paměti

U tohoto bufferu je důležité zmínit dvě věci. Za prvé je nutné pro buffer nastavit parametr VK_BUFFER_USAGE_TRANSFER_SRC_BIT, který říká, že tento buffer bude využit jako zdroj pro pozdější transfer dat. Za druhé je nutné nastavit parametr VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT, který umožní, aby se dal tento buffer dále zvenčí namapovat pomocí VkMapMemory.

Po vytvoření vyrovnávacího bufferu přichází na řadu namapování paměti v oblasti bufferu a přímé zkopírování pixelů. Pro namapování je potřeba znát tyto parametry.

- VkDevice – jednotka, na které dochází k dané operaci
- VkDeviceMemory – paměť bufferu, která bude namapována
- offset – offset od začátku namapovaného objektu v paměti
- VkDeviceSize – velikost mapované paměti

- VkMemoryFlags – aktuálně pouze rezervováno pro budoucí využití
- ppData – ukazuje na pointer, ve kterém je vrácen další pointer, na který lze přistoupit zvenčí a který ukazuje na začátek mapovaného rozsahu dat pro kopírování v paměti

Následně je použita funkce memcpy, díky které dojde k přímému zkopírování načtených dat textury do vyrovnávacího bufferu. Poté je zavoláná funkce VkUnmapMemory, která již podle svého názvu odmapuje použitou paměť.

5.2.3.1 Vytvoření objektu pro texturu

Díky pomocné funkci createImage nyní dojde k vytvoření objektu pro nový obrázek textury. Opět je nutné explicitně vyjádřit nastavení některých parametrů daného obrázku pro strukturu VkImageCreateInfo. Jedná se o předání informací o výšce a šířce, o jaký typ obrázku se jedná (např.: VK_IMAGE_TYPE_2D), formátu obrázku (např.: VK_FORMAT_R8G8B8A8_UNORM), tiling obrazu (VK_IMAGE_TILING_LINEAR nebo VK_IMAGE_TILING_LINEAR). Dále typ použití (např.: TRANSFER_DST_BIT) a řešení paměti (např.: VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT). Další parametry jsou volitelné a není nutné je vyplňovat, pokud to není nutné.

5.2.3.2 Vytvoření command bufferu

Na tomto místě se poprvé potkáváme s vytvořením command bufferu. Je potřeba vytvořit samotný objekt command bufferu, alokovat a ujasnit strukturu VkCommandBufferAllocateInfo, nastartovat nahrávání vytvořeného command bufferu pomocí vkBeginCommandBuffer() a provést požadovanou operaci (v tomto případě změnu uspořádání obrazu). Ukázkový kód níže.

```
insertImageMemoryBarrier(copyCmd, textureImage, VK_ACCESS_TRANSFER_WRITE_BIT,
    VK_ACCESS_SHADER_READ_BIT, VK_IMAGE_LAYOUT_UNDEFINED,
    VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL, VK_PIPELINE_STAGE_TRANSFER_BIT,
    VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT, VkImageSubresourceRange{
    VK_IMAGE_ASPECT_COLOR_BIT, 0, 1, 0, 1 });
```

Výpis 8: Ukázka vytvoření command bufferu pro texturu.

Nyní k popisu jednotlivých parametrů:

- copyCmd – použitý command buffer
- textureImage – objekt obrázku, který bude použit
- VK_ACCESS_TRANSFER_WRITE_BIT – vstupní přístupová maska
- VK_ACCESS_SHADER_READ_BIT – výstupní přístupová maska
- VK_IMAGE_LAYOUT_UNDEFINED – vstupní uspořádání
- VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL – výstupní uspořádání

- VK_PIPELINE_STAGE_TRANSFER_BIT – vstupní bitová maska
- VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT – výstupní bitová maska
- VkImageSubresourceRange – bitová maska specifikující aspekty obrazu a další nastavení

Následně ukončíme nahrávání pomocí vkEndCommandBuffer() a odešleme nahraný command buffer ke zpracování pomocí submitWork(). Změnu uspořádání obrazu je potřeba udělat proto, aby bylo možné zkopírovat vyrovnávací buffer do objektu vkImage, jež je součástí mé struktury ModelTexture.

U změny uspořádání obrazu ve Vulkan API se na chvíli zastavím. Při práci s VkImage je vždy nutné explicitně říci, jaké uspořádání bude obraz mít, protože každý typ uspořádání je použit na něco jiného. Vulkan API obsahuje aktuálně deset různých uspořádání obrazu:

- VK_IMAGE_LAYOUT_UNDEFINED
- VK_IMAGE_LAYOUT_GENERAL
- VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL
- VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL
- VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL
- VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL
- VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL
- VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL
- VK_IMAGE_LAYOUT_PREINITIALIZED
- VK_IMAGE_LAYOUT_PRESENT_SRC_KHR

Například VK_LAYOUT_UNDEFINED lze použít pouze jako vstupní uspořádání. K tomuto uspořádání nemají přístup žádné zařízení a nelze z něj provést například operaci kopírování. Naproti tomu uspořádání VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL už dle názvu říká, že je možné jej použít pouze pro přenos dat ze zdrojového obrazu. Na příkladu uvedeném výše dochází k přechodu z uspořádání VK_IMAGE_LAYOUT_UNDEFINED na VK_IMAGE_LAYOUT_DST_OPTIMAL. Po vysvětlení výše už by mělo být patrné, proč se tak stalo. Jelikož potřebujeme zkopírovat obsah vyrovnávacího bufferu do obrazu textury, je potřeba tento obraz připravit, aby data mohl přijímat (VK_IMAGE_LAYOUT_DST_OPTIMAL).

5.2.3.3 Kopírování vyrovnávacího bufferu do obrazu Jakmile je obraz připraven přijmou data z vyrovnávacího bufferu, je možné provést samotné zkopírování dat díky pomocné funkci copyBufferToImage. Tato funkce si v parametrech převezme objekt vyrovnávacího bufferu, objekt obrazu, jeho šířku a výšku.

Funkce copyBufferToImage je v zásadě velmi jednoduchá. Je opět potřeba vytvořit a alokovat command buffer a následně začít nahrávat požadovanou operaci. V případě kopírování Vulkan API explicitně řekneme, jak budeme kopírovat a vyplníme strukturu VkBufferImageCopy. V této struktuře nastavíme vše potřebné (např.: offset, bitovou masku, šířku a výšku). Následuje zavolání funkce vkCmdCopyBufferToImage, která ve svých parametrech odešle informace

o command bufferu, kopírovaném vyrovnávacím bufferu, obrazu určeného pro texturu, výslednou podobu uspořádání a informace o vyplněné struktuře VkBufferImageCopy. Poté ukončíme nahrávání command bufferu a celý nahraný command buffer odešleme ke zpracování do fronty.

5.2.3.4 Příprava obrazu textury pro fragment shader Jelikož je nyní obraz textury v uspořádání vhodném pro kopírování dat, musí dojít k ještě jedné změně uspořádání, jinak by nebylo možné, aby k tomuto obrazu textury přistoupil fragment shader. O shaderech samotných bude vedena další kapitola. Znovu spustíme nahrávání command bufferu a opět je použita funkce insertImageMemoryBarrier, parametry jsou však jiné. Z přístupové masky VK_ACCESS_TRANSFER_WRITE_BIT dojde k přechodu na VK_ACCESS_SHADER_READ_BIT.

Dále dojde ke změně již zmíněného uspořádání z VK_IMAGE_LAYOUT_DST_OPTIMAL na VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL. Nakonec dojde také na bitové masky, kde proběhne změna z VK_PIPELINE_STAGE_TRANSFER_BIT na VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT. Následně ukončíme nahrávání command bufferu a předáme command buffer do fronty ke zpracování. Výsledný obraz textury je nyní v uspořádání vhodném pro čtení fragment shaderem.

Obrázek 2: Ukázka načteného modelu včetně textur.

5.2.4 Nastavení TextureView

Aby bylo možné s VkImage obrazy pracovat, musí mít každý takový obraz svůj VkImageView. Jedná se doslova o „pohled“ na na tyto obrazy. VkImageView popisuje, jakým způsobem se dá k obrazu přistoupit nebo také ke které části obrazu přistoupit. Například zda se jedná o texturu, nebo hloubkovou mapu.

VkImageView je vytvořeno pomocnou funkcí createImageView, která pracuje s dvěma parametry. Jedná se o objekt samotného obrazu a následně formát, který má být použit. Vulkan API obsahuje několik desítek různých formátu, se kterými obrazy mohou pracovat. Namátkově například VK_FORMAT_UNDEFINED, VK_FORMAT_B8G8R8A8_UNORM, VK_FORMAT_R64G64B64A64_SFLOAT. Formát je vždy volen s ohledem na další použití. Předtím, než je možné vytvořit VkImageView je nutné nachystat strukturu VkImageViewCreateInfo. Této struktuře je mimo jiné nutné předat informace o objektu obrazu a jeho formátu, dále nastavit typ a bitovou masku (např.: VK_IMAGE_ASPECT_COLOR_BIT). Pomocí funkce vkCreateImageView dojde k vytovření VkImageView pro obraz textury.

5.2.5 Nastavení TextureSampler

Sampler slouží k tomu, aby poskytl informace o tom, jak budou obrazové data čtená uvnitř shaderů. Oproti OpenGL, kde při vytvoření textury dochází jak k vytvoření obrazu tak sampleru najednou, je nutné ve Vulkan API obě věci specifikovat zvlášť. Obraz již máme vytvořený a nyní je potřeba vytvořit sampler. Při tvorbě sampleru dochází k nastavení povolení či zakázání filtrování textury, zda chceme používat mipmapy (nebo jen konkrétní rozsah) a také jaký typ adresování chceme použít (clamp vs wrap). Vytvoření samotného sampleru probíhá opět v pomocné funkci, kde je nejprve potřeba připravit strukturu VkSamplerCreateInfo. Příklad nastavení sampleru může vypadat takto:

```
VkSamplerCreateInfo samplerInfo = {};
samplerInfo.sType = VK_STRUCTURE_TYPE_SAMPLER_CREATE_INFO;
samplerInfo.magFilter = VK_FILTER_LINEAR;
samplerInfo.minFilter = VK_FILTER_LINEAR;
samplerInfo.addressModeU = VK_SAMPLER_ADDRESS_MODE_REPEAT;
samplerInfo.addressModeV = VK_SAMPLER_ADDRESS_MODE_REPEAT;
samplerInfo.addressModeW = VK_SAMPLER_ADDRESS_MODE_REPEAT;
samplerInfo.anisotropyEnable = VK_TRUE;
samplerInfo.maxAnisotropy = 16;
samplerInfo.borderColor = VK_BORDER_COLOR_INT_OPAQUE_BLACK;
samplerInfo.unnormalizedCoordinates = VK_FALSE;
samplerInfo.compareEnable = VK_FALSE;
samplerInfo.compareOp = VK_COMPARE_OP_ALWAYS;
samplerInfo.mipmapMode = VK_SAMPLER_MIPMAP_MODE_LINEAR;
```

Výpis 9: Ukázka nastavení struktury VkSamplerCreateInfo pro textureSampler.

Nakonec zbývá zavolat funkci vkCreateSampler, která se postará o vytvoření sampleru.

5.2.6 Uložení vytvořené textury

Jakmile proběhnou všechny tři kroky, tedy createTextureImage, createTextureView a createTextureSampler, zbývá už pouze jediná věc – uložení vytvořeného obrazu textury. Ve své aplikaci texturu uložím do kontejneru typu vektor. Zde pak po celou dobu běhu aplikace skladuji všechny textury, které byly pro model vytvořeny.

5.2.7 Sestavení modelu

Poté, co jsou ve svých vektorech načteny informace o materiálech i texturách, přichází na řadu sestavení celého modelu. Postupnou iterací skrze všechny části načteného modelu dochází k naplněný vektorů s vrcholy a indexy. U vrcholů dochází k zaznamenání polohy, souřadnic pro textury, normál a barvy.

5.2.8 Unikátní vrcholy

Je běžné, že 3D modely obsahují duplicitní vrcholy a díky tomu se také zpomaluje načítání a vykreslování jednotlivých modelů. Tyto duplicity vznikají z toho důvodu, že některé vrcholy jsou obsaženy ve více různých trojúhelnících, které jsou postupně procházeny. Na řadu zde přichází optimalizace v podobě index bufferu a již zmíněného vektoru s indexy. Pro účely optimalizace je zde implementován kontejner std::unordered_map.

```
std::unordered_map<Vertex, uint32_t> uniqueVertices = {};
```

Výpis 10: Ukázka vytvoření kontejneru pro unikátní vrcholy.

V tom budou uloženy pouze unikátní vertexy, které budou pomocí index bufferu znovu použity maximálním možným způsobem. Při načítání modelu a jeho vrcholů je u každého přečteného vrcholu zkontrolováno, jestli se stejný vrchol už jednou neobjevil (vrchol musí mít naprosto stejnou pozici a souřadnice textury). Pokud takový vrchol neexistuje, pak jej vložíme do vektoru vrcholů a index tohoto vrcholu uložíme do výše zmíněného kontejneru uniqueVertices. Následně dojde k uložení tohoto indexu také do vektoru indexů. Pokud nastane případ, kdy je nalezen duplicitní vrchol se stejnými parametry, pak nejprve najdeme jeho index v kontejneru uniqueVertices a tento index uložíme také do vektoru indexů. Díky tomu, že je u kontejneru uniqueVertices použit jako klíč pro hashovací tabulku mnou definovaná strukturu Vertex, je potřeba doplnit ještě dvě funkce. Jedná se o test rovnosti a výpočet hash funkce. První funkci je možné vyřešit přepsáním operátoru „==“ ve struktuře Vertex v následujícím tvaru.

```
bool operator==(const Vertex& other) const {
return pos == other.pos && color == other.color && texCoord == other.texCoord
    && normal == other.normal;
}
```

Výpis 11: Ukázka úpravy operátoru pro test rovnosti.

Díky tomuto přepsání funkce operátoru „==“ dochází při porovnávání vrcholů ke kontrole již zmíněné pozice, barvy, souřadnic textur a normál vrcholu. Hashovací funkce je pak implementována pomocí template standardní knihovny (std::hash<T>). Jelikož je v hashovací funkci použitý typ proměnné z knihovny GLM, je nutné z této knihovny nadefinovat makro #define GLM_ENABLE_EXPERIMENTAL a také připojit soubor <glm/gtx/hash.hpp>.

U modelu už zbývá zmínit pouze poslední funkci a to recordIndexInfo. Tato funkce přebírá celkem tři parametry.

- indexCount – počet indexů, které budou v rámci jedné části modelu vykresleny
- indexStart – čísla indexu, od kterého začne vykreslování části modelu
- textureStart + currentMat – index pro pole s texturami, který je později předávám fragment shaderu

Tato funkce je zavolána u každé části načteného modelu. Později je díky této metodě jasné, co je potřeba v rámci volání vykreslit.

5.2.9 Příprava vertex a index bufferů

Buffer ve Vulkánu je v podstatě oblast paměti, která je použita pro ukládání libovolných dat, které může následně přečíst grafická karta. V této části dojde k vytvoření bufferů pro vrcholy a indexy načteného modelu, nicméně buffery jako takové mohou být ve Vulkanu použité pro mnoho dalších účelů. Jelikož tyto buffery pro sebe nedokážou automaticky alokovat paměť, je vždy nutné pamět alokovat předem. Naštěstí pro programátora je to snadná operace, viz. níže.

```
const VkDeviceSize vertexBufferSize = vertices.size() * sizeof(Vertex);
const VkDeviceSize indexBufferSize = indices.size() * sizeof(uint32_t);
```

Výpis 12: Ukázka alokace paměti pro vertex a index buffer.

Pomocí těchto dvou řádků jsem alokoval dostatečné množství paměti, které je určeno dle velikosti vektoru vrcholů, respektive indexů. I v této kapitole je pro oba nově vytvořené buffery použit tzv. vyrovnávací buffer. Tvorba bufferů pokračuje již vysvětleným vytvořením, alokováním a spuštěním command bufferu a následně vytvořením samotných bufferů díky pomocné funkci createBuffer. Rozdílem v této části je pak kopírování jednoho bufferu do druhého, tedy kopírování z vyrovnávacího bufferu do bufferu vrcholů, respektive do bufferu indexů pomocí funkce vkCmdCopyBuffer. Tato funkce má celkem pět parametrů.

- objekt command bufferu
- vyrovnávací buffer
- výsledný buffer
- počet oblastí bufferu ke kopírování (obyčejně 1)

- VkBufferCopy struktura

Poté je command buffer opět odeslán ke zpracování do fronty. Stejná práce proběhne také u index bufferu. U bufferu vrcholů je nutné explicitně nastavit typ použití na hodnotu VK_BUFFER_USAGE_VERTEX_BUFFER_BIT, u bufferu indexů pak na hodnotu VK_BUFFER_USAGE_INDEX_BUFFER_BIT. Tím je práce na vytvoření těchto dvou bufferů hotova.

5.2.10 Vytvoření uniform bufferu

Uniform buffer obsahuje transformační a projekční matice, ke kterým skrze Vulkan deskriptory přistupuje vertex shader. O Vulkan dekriptorech i vertex shaderu se rozepíši v pozdější kapitole. Struktura objektu uniform bufferu v mé aplikace je znázorněna níže.

```
struct UniformBufferObject {
   glm::mat4 model;
   glm::mat4 view;
   glm::mat4 proj;
   glm::vec3 lightPosition;
};
```

Výpis 13: Ukázka struktury Uniform bufferu v aplikaci.

Jedná se o známou strukturu, kterou používá velká většina grafických aplikací a nejedná se tedy o nic nového. Vytvoření samotného uniform bufferu není nic složitého. Postup je stejný, jako v případě předchozích dvou bufferů pro vrcholy a indexy. Za prvé je potřeba alokovat dostatečné množství paměti. Ukázka alokace paměti pro uniform buffer je přiložena níže.

```
VkDeviceSize bufferSize = sizeof(UniformBufferObject);
```

Výpis 14: Ukázka alokace paměti pro Uniform buffer.

Poté je opět zavolána funkce createBuffer s tím, že jako typ použití je potřeba vyplnit VK_BUFFER_USAGE_UNIFORM_BUFFER_BIT.

V aplikacích, které jsou dynamické a probíhá v nich neustálé vykreslování na obrazovku, obyčejně existují další funkce, které při každém vypočteném snímku aktualizují uniform buffer. V případě mé aplikace není toto potřeba, protože jednak nic nevykreslujeme na obrazovku a jednak je cílem pouze statický obrázek. Díky tomu je zde uniform buffer značně zjednodušen.

5.3 Framebuffer, renderpass a descriptor sety

V této části se podíváme na přípravu obrazů pro framebuffer a jeho samotné vytvoření. Dále vytvoříme renderpass a také descriptor sety, jenž jsou nutné pro práci s shadery.

5.3.1 Vytvoření obrazových příloh pro framebuffer

Framebuffer je objekt, který v sobě drží referenci na všechny VkImageView objekty, které reprezentují přílohy tohoto framebufferu. Přílohou je myšlena část framebufferu odpovídající za konkrétní informace. V mé aplikace využívám dva typy příloh. Klasickou obrazovou přílohu a pak také hloubkovou mapu pro zjištění vzdálenosti modelu od kamery. Abych tyto dvě přílohy mohl používat, je potřeba je pro framebuffer nejdříve připravit.

Předtím, než dojde k nastavení jednotlivých struktur pro obě přílohy framebufferu je nutné definovat, jakou budou mít šířku a výšku. V mém případě je také nutné zjistit, jaký hloubkový formát budu schopen využít, jelikož každá grafická karta podporuje jiný typ. K tomuto účelu je využita funkce getSupportedDepthFormat, která přebírá dva parametry a to použitou fyzickou jednotku a také pointer na hloubkový formát. Vzhledem k tomu, že jakýkoliv hloubkový formát a jeho podpora je dobrovolná a založena pouze na vůli daného výrobce grafické karty, procházím jednotlivé formáty od toho nejpřesnějšího až k tomu nejméně přesnému a zjišťuji, který mohu využít. Funkce následně vrátí první nalezený hloubkový formát, který je možné využít.

V další části je potřeba opět vytvořit již jednou vysvětlené v části tvorby textur. Ve zkratce se jedná o vytvoření a naplnění strukturu pro oba obrazy (VkImageCreateInfo) a dále vytvoření VkImageView.

Důležité je pohlídat nastavení první přílohy jako VK_IMAGE_ASPECT_COLOR_BIT. Příloha pro hloubku pak bude nastavena jako VK_IMAGE_ASPECT_DEPTH_BIT | VK_IMAGE_ASPECT_STENCIL_BIT.

5.3.2 Vytvoření renderpassu

Jedná se o navazující funkci hned za vytvořením příloh pro framebuffer. Renderpass je místo, kde Vulkan API řekneme, které přílohy budou při zpracování použity, kolik budeme používat příloh pro barvy a hloubku nebo jak se bude pracovat s jejich obsahem při vykreslování. V mém případě využíváme v renderpassu dvě přílohy, viz. předchozí kapitola. První příloha se stará o barvy a druhá příloha o hloubku. Díky tomu, že využívám více než jednu přílohu, je již nutné vytvořit pole, ve kterém budou tyto přílohy popsány.

```
std::array<VkAttachmentDescription, 2> attchmentDescriptions = {};
```

Výpis 15: Ukázka inicializace struktury VkAttachmentDescription pro renderpass.

Následně je nutné pro každou přílohu explicitně nastavit správné informace pro tuto strukturu.

- format – formát, jaký je pro obraz použit
- samples – určuje počet vzorků na jeden pixel v obraze
- loadOp – jakým způsobem bude využit obsah obrazu při započetí renderpassu
- storeOp – co se bude dít s obsahem obrazu po ukončení renderpassu

- stencilLoadOp – jakým způsobem bude využit obsah komponenty při započetí renderpassu
- stencilStoreOp – co se bude dít s obsahem obrazu po ukončení renderpassu
- vstupní uspořádání obrazu při započení renderpassu
- finální uspořádání obrazu po ukončení renderpassu

Pro každou z těchto příloh je pro strukturu VkSubpassDecription potřeba předat referenci, aby s nimi dokázal renderpass pracovat.

```
VkAttachmentReference colorReference = { 0,
    VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL };
VkAttachmentReference depthReference = { 1,
    VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL };
```

Výpis 16: Ukázka nastavení referencí renderpassu.

Před vytvořením samotného renderpassu je potřeba vykonat poslední krok a to změnu uspořádání obrazů, bitových mask a zavislostí tak, aby byly připraveny pro práci s renderpassem.

Následně dojde k vytvoření renderpassu. Před zavoláním finální funkce je - jak už je ve Vulkanu zvykem - nutné vytvořit a vyplnit strukturu VkRenderPassCreateInfo. Po nastavení této struktury zavoláme funkci vkCreateRenderPass se čtyřmi parametry (logická jednotka, pointer na vytvořenou renderpass strukturu, null pointer pro alokaci callbacku a pointer na objekt renderpassu), která renderpass konečně vytvoří.

Po vytvoření renderpassu je nyní možné přejit k vytvoření dvou pohledů VkImageViews. Jedná se o VkImageView, které ukazují na přílohy vytvořené pro framebuffer v předchozí kapitole. Jakmile máme tyto pohledy hotové, můžeme finálně začít vytvářet framebuffer. Nejprve vytvoříme a naplníme strukturu VkFramebufferCreateInfo. Níže popíši strukturu pro vytvoření framebufferu.

- sType – jasně daný typ této struktury
- renderPass – objekt renderpassu, který byl právě vytvořen
- attachmentCount – počet příloh ve framebufferu
- width – šířka framebufferu
- height – výška framebufferu
- layers – počet vrstev framebufferu

Zbývá už jen poslední krok a to zavolat funkci vkCreateFrameBuffer, která následně se čtyřmi parametry (logická jednotka, pointer na založenou strukturu framebufferu, null pointer pro alokaci callbacku a pointer na objekt framebufferu) vytvoří framebuffer.

5.3.3 Descriptor sety

V této části začneme úvodem do kompletního nastavení decriptor setů, jenž se skládá z vytvoření descriptor set layoutu, descriptor set poolu a samotného decriptor setu.

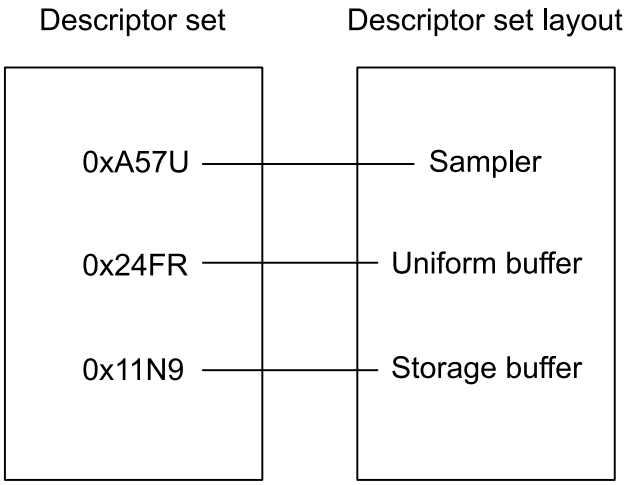
Zjednodušeně řečeno jsou descriptor sety určeny k tomu, abychom shaderům řekli, jak mají pracovat s daty které v aplikaci poskytujeme (Vkbuffer, VkImage textury atd.). Descriptor sety jsou v podstatě kontejnerem, který drží o těchto datech informace. Každá data, která chceme předávat shaderům, musí být v descriptor setech přesně popsána a opět k tomu slouží jasně definované struktury. Do každého descriptor setu je možné vložit více různých dat i jejich typů, ale je nutné data přesně specifikovat. V shaderech následně probíhá podobná specifikace. Dohromady pak tvoří jakýsi typ rozhraní, který spolu komunikuje. Jednotlivé data uvnitř descriptor setu se nazývají descriptory. Vulkan má celkem jedenáct typů descriptorů. Jsou to:

- samplers – definují, jak budou čtena obrazová data
- sampled images – definují obrazy, ze kterých v shaderech můžeme číst
- combined image samplers – kombinuje samplery a sampled images do jednoho objektu
- storage images – umožňuje jak čtení obrazových dat, tak také jejich zápis do obrazu
- input attachments – specifické využití pro renderpass
- uniform buffers – umožňuje číst data z proměných, které poskytuje uniform buffer
- storage buffers – umožňuje číst i ukládat data z a do proměnných
- uniform texel buffers – umožní pracovat s obsahem bufferu jako kdyby obsahoval obrovské množství texturových dat (větší než uniform buffer)
- storage texel buffers - podobné jako předchozí bod, ale navíc mohou data mimo čtení také zapisovat

Všechny výše zmíněné descriptory se dají vytvořit buď z bufferů, obrazů nebo samplerů. Rozdíl pak vzniká hlavně v tom, jakým způsobem je používáme uvnitř shaderů. V mé aplikace jsem použil combined image sampler a nabízí se otázka, proč není použit například uniform texel buffer, který se zdá být pro textury vhodnější. Jelikož má aplikace načítá pouze jeden model s maximálně několika desítkami textur, není potřeba využít přímo uniform texel buffer. Tento buffer je totiž optimalizovaný pro obrovské množství texturových dat a to má aplikace nepotřebuje. Lepšího výsledku tedy dosáhneme pomocí combined image sampleru.

5.3.3.1 Descriptor set layout Abychom mohli připravit pro shadery potřebná data, musíme začít vytvořením descriptor layoutu. Descriptor layout definuje strukturu descriptor setu, konkrétně jaký typ descriptorů bude použit, počet descriptorů určitého typu a také jejich pořadí.

Vytvoření layoutu začíná definováním parametrů ve struktuře VkDescriptorSetLayoutBinding. V mé aplikace tuto strukturu používám celkem třikrát. Jednou pro Uniform buffer, po-



Obrázek 3: Vazba mezi descriptor sety a layouty ve Vulkan API. V descriptor setech ukládáme informace o zdrojích se kterými pracujeme. Descriptor layout pak definuje strukturu deskriptor setů - tedy typ obsažených descriptorů, jejich počet a pořadí.

druhé pro práci s texturami a potřetí k vytvoření pole, které bude pracovat s předchozími dvěma strukturami. Příklad struktury pro descriptor layout uniform bufferu níže.

```
VkDescriptorSetLayoutBinding uboLayoutBinding = {};
uboLayoutBinding.binding = 0;
uboLayoutBinding.descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
uboLayoutBinding.descriptorCount = 1;
uboLayoutBinding.stageFlags = VK_SHADER_STAGE_VERTEX_BIT;
```

Výpis 17: Ukázka descriptor layoutu pro uniform buffer.

Struktura opět obsahuje několik různých parametrů, které v následujícím seznamu vysvětlím.

- binding – je index použitého descriptoru v descriptor setu, který je stejný také pro shader
- descriptorType – typ použitého descriptoru
- descriptorCount – počet descriptorů stejného typu, ke kterému je přistupováno jako k poli
- stageFlags – specifikuje typ shaderu, který bude mít k danému descriptoru přístup

Nyní vytvoříme pole pro strukturu VkDescriptorSetLayoutBinding, do kterého vložíme již vytvoření struktury pro Uniform buffer a combined sampler. K této koncové struktuře vytvoříme také další strukturu VkDescriptorSetLayoutCreateInfo, která je potřeba pro samotné vytvoření descriptor layoutu. V této struktuře explicitně vyjádříme používání obou vytvořených descriptorů a také pointer na pole s popisem jednotlivých descriptorů. Jako poslední zavoláme

funkci vkCreateDescriptorSetLayout, která přebírá čtyři parametry (logickou jednotku, strukturu VkDescriptorSetLayoutCreateInfo, nullpointer na callback API a pointer na hlavní objekt descriptor layoutu). Pokud je vše nastaveno správně, pak dojde k vytvoření descriptor layoutu a můžeme se přesunout k vytvoření descriptor poolu.

5.3.3.2 Descriptor pool Abychom v dalším kroku mohli vytvořit descriptor set, musíme mimo descriptor layoutu vytvořit také descriptor pool. Descriptor sety samy o sobě (stejně jako třeba command buffery) nejsou vytvářeny na přímo, ale jsou alokovány právě z descriptor poolu.

K vytvoření descriptor poolu musíme specifikovat tyto informace:

- kolik descriptor setů může být z poolu alokováno
- počet a typ descriptorů, které mohou být z descriptor poolu akolovány napříč všemi descriptor sety

Nejdříve je potřeba vytvořit a naplnit strukturu VkDescriptorPoolSize. Díky předchozí kapitole víme, že v mé aplikaci se tato struktura chová jako pole o velikosti 2. Do této struktury jsou následně vloženy dva descriptory vytvořené v předchozím kroku, viz.

```
std::array<VkDescriptorPoolSize, 2> poolSizes = {};
poolSizes[0].type = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
poolSizes[0].descriptorCount = modelTextures.size();
poolSizes[1].type = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
poolSizes[1].descriptorCount = modelTextures.size();
```

Výpis 18: Ukázka vytvoření descriptor poolu.

V tomto bodě specifikujeme typ descriptoru a jeho celkový maximální počet, který bude alokován z poolu. Poté vytvoříme další strukturu VkDescriptorPoolCreateInfo, kterou naplníme daty z předchozí vytvořené struktury VkDescriptorPoolSize.

Následně zavoláme funkci vkCreateDescriptorPool se čtyřmi parametry (logická jednotka, pointer na VkDescriptorPoolCreateInfo, nullpointer na callback API a pointer na descriptor poolu). Zbývá vyřešit vytvoření descriptor setu, který popíšu v další kapitole.

5.3.3.3 Vytvoření descriptor setu Před námi je poslední část týkající se descriptorů – vytvoření descriptor setu. Pro alokaci descriptor setu je potřeba vytvořit ještě poslední strukturu a to VkDescriptorSetAllocateInfo. V této struktuře předáme informace z již vytvořeného descriptor layoutu a descriptor poolu a můžeme zavolat funkci vkAllocateDescriptorSets. Tato funkce přebírá tři parametry (logickou jednotku, strukturu VkDescriptorSetAllocateInfo a pointer na objekt descriptor setu).

Po vytvoření je descriptor set prázdný. Nyní je potřeba jej nastavit, případně aktualizovat. V tomto místě ovladači grafické karty řekneme, jaká data budou použita pro descriptory uvnitř

descriptor setu. Jelikož používáme dva descriptory, i zde je nutné vytvořit pole pro strukturu VkWriteDescriptorSet. Nastavení pro zápis do descriptor setu vypadá kupříkladu takto:

```
descriptorWrites[0].sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
descriptorWrites[0].dstSet = descriptorSet;
descriptorWrites[0].dstBinding = 0;
descriptorWrites[0].dstArrayElement = 0;
descriptorWrites[0].descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
descriptorWrites[0].descriptorCount = 1;
descriptorWrites[0].pBufferInfo = &bufferInfo;
```

Výpis 19: Ukázka nastavení zápisu pro descriptor sety.

Opět se jedná o jasně danou strukturu, kterou je nutné explicitně nastavit. V seznamu níže vysvětlím jednotlivé parametry.

- sType – jasně daný typ struktury
- dstSet – objekt descriptor setu, který chceme aktualizovat
- dstBinding – index descriptor setu, který chceme aktualizovat
- dstArrayElement – první index v poli, který chceme aktualizovat
- descriptorType – typ použitého descriptoru
- descriptorCount – počet descriptorů, které chceme aktualizovat
- pBufferInfo – pointer na uniform buffer

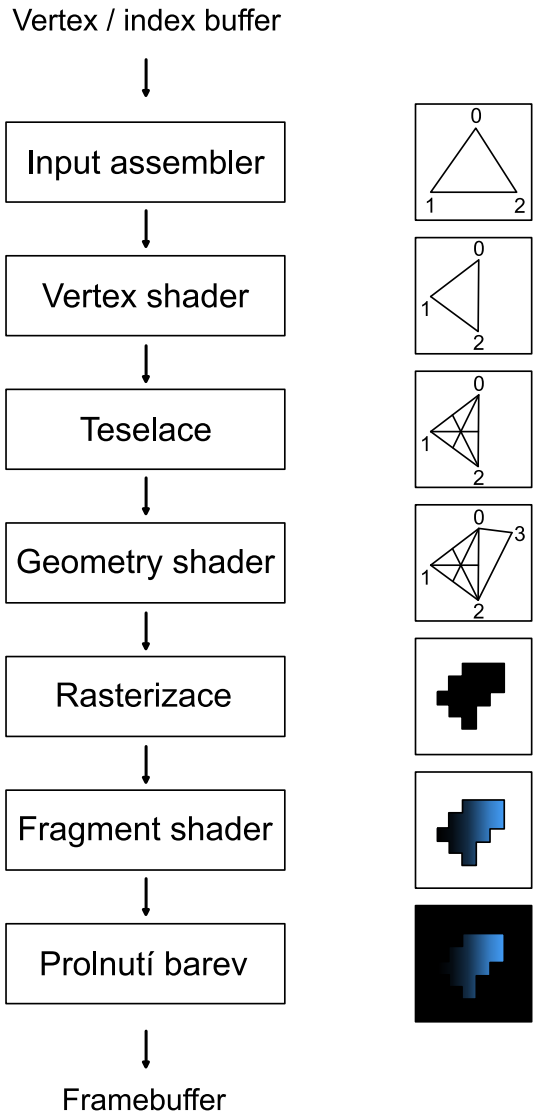
Zbývá už jen zavolat potřebnou funkci vkUpdateDescriptorSets se čtyřmi parametry (logická jednotka, velikost dle VkWriteDescriptorSet, data dle VkWriteDescriptorSet, velikost pDescriptorCopies, nullpointer na VkCopyDescriptorSet).

Zaktualizovali jsme descriptor set a dostali se na konec funkce pro vytvoření descriptor setu. S descriptor sety nicméně nejsme ještě úplně u konce. V další kapitole, která se týká grafické pipeline Vulkan API, se descriptor setů ještě dotkneme.

5.4 Grafická pipeline

V této kapitole se dle mého názoru dostáváme do jedné z nejdůležitějších částí Vulkan API. Grafická pipeline zajišťuje to, že se nám vůbec na konci podaří něco vykreslit. Jedná se o sekvenci operací, díky kterým se z počátečních vrcholů a textur načteného modelu podaří vykreslit jednotlivé pixely na finální obraz.

Grafická pipeline má několik částí, ze kterých se skládá. Jedná se o input assembler, vertex shader, tesselaci, geometry shader, rasterizaci, fragment shader, prolnutí barev. Jednotlivé části postupně projdu. Celé nastavení grafické pipeline je uloženo ve struktuře VkGraphicsPipelineCreateInfo a může vypadat například takto:



Obrázek 4: Průběh grafické pipeline. Grafická pipeline má několik částí, ze kterých se skládá. Jedná se o input assembler, vertex shader, tesselaci, geometry shader, rasterizaci, fragment shader, prolnutí barev.

```
std::array<VkPipelineShaderStageCreateInfo, 2> shaderStages{};

pipelineCreateInfo.pInputAssemblyState = &inputAssemblyState;
pipelineCreateInfo.pRasterizationState = &rasterizationState;
pipelineCreateInfo.pColorBlendState = &colorBlendState;
pipelineCreateInfo.pMultisampleState = &multisampleState;
pipelineCreateInfo.pViewportState = &viewportState;
```

```
pipelineCreateInfo.pDepthStencilState = &depthStencilState;
pipelineCreateInfo.pDynamicState = &dynamicState;
pipelineCreateInfo.stageCount = static_cast<uint32_t>(shaderStages.size());
pipelineCreateInfo.pStages = shaderStages.data();
```

Výpis 20: Ukázka struktury grafické pipeline.

5.4.1 Input assembler

Tato část se stará o první data. Jedná se o data z vertex bufferu a v našem případě také index bufferu, aby nebylo nutné některé vrcholy neustále dokola opakovat.

Za tuto část odpovídá struktura VkPipelineInputAssemblyStateCreateInfo, ve které je důležité nastavit, jaký typ geometrie bude z vrcholů vykreslován. V našem případě použiji VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST, kde už z názvu lze poznat, že půjde o klasické trojúhelníky.

5.4.2 Vertex Shader

Vertex shader následně prochází každý vrchol, který byl input assemblerem zpracován a aplikuje transformační matice, aby došlo ke změně souřadnic z modelového prostoru do 3D prostoru aplikace.

V této části je nutné nastavit dvě struktury. Jedná se o VkVertexInputBindingDescription a VkVertexInputAttributeDescription. Tyto struktury je potřeba k tomu, abychom řekli Vulkan API, jakým způsobem má vertex data předat shaderům v momentě, kdy už má tato data v paměti grafická karta. První struktura zodpovídá za nastavení, které popisuje, jakým způsobem budou načteny data z paměti. Zde jsou pouze dvě možnosti a to VK_VERTEX_INPUT_RATE_VERTEX nebo VK_VERTEX_INPUT_RATE_INSTANCE. První možnost znamená, že se v paměti posouváme vždy po každém vrcholu, druhá pak posun po každé instanci. Druhá možnost se nicméně používá hlavně pro instancované renderování, které v mé práci nepoužívám.

Struktura VkVertexInputAttributeDescription popisuje, jakým způsobem s načtenými daty pracovat a kde se v načtených datech nachází různé parametry. Vypadá například takto:

```
std::vector<VkVertexInputAttributeDescription> vertexInputAttributes = {
   vertexInputAttributeDescription(0, 0, vkFormat, offsetof(Vertex, pos)),
   vertexInputAttributeDescription(0, 1, vkFormat, offsetof(Vertex, color)),
   vertexInputAttributeDescription(0, 2, vkFormat, offsetof(Vertex, texCoord)),
   vertexInputAttributeDescription(0, 3, vkFormat, offsetof(Vertex, normal)),
};
```

Výpis 21: Ukázka struktury input atributů pro vertex shader.

Jedná se v podstatě o pole jednotlivých částí vertexu, kde je nutné určit čtyři parametry – binding, location, format a offset. Binding je číslo atributu vertexu, odkud jsou brány data. Location je číslo binding location v shaderu pro daný atribut. Format je velikost a typ atributu a offset je číslo poukazující na začátek jednotlivých hodnot atributů vertexu.

5.4.3 Teselace

Teselace slouží ke zkvalitnění načítaných modelů pomocí rozdělení jednotlivých částí modelu na ještě menší části. Tento modul má i negativní efekt, kterým je samozřejmě větší náročnost na vykreslování. Ve Vulkan API je tento modul dobrovolný a v mé aplikaci jej nenajdete, jelikož jej pro účely aplikace není potřeba.

5.4.4 Geometry shader

Geometry shader se spouští na jakékoliv jednoduché geometrii (např.: trojúhelník, bod) a buď může tuto geometrii uplně zrušit a nebo počty dané geometrie ještě znásobit. Opět se jedná o dobrovolný modul a v mé aplikace jej nenajdeme, jelikož nemá praktické využití.

5.4.5 Rasterizace

V této fázi se z jednoduchých těles stávají fragmenty. Fragmenty jsou v zásadě pixelové prvky mířící do framebufferu. Všechny fragmenty, které jsou mimo plochu framebufferu jsou odstraněny. Následně dojde k interpolaci těchto fragmentů s výsledky z vertex shaderu. Při rasterizaci také dochází k testování hloubky jednotlivých pixelů, „culling“ jednotllivých vykreslovaných ploch a také „nůžkový“ test.

Za nastavení rasterizátoru odpovídá ve Vulkan API struktura VkPipelineRasterizationStateCreateInfo. V této struktuře probíhá nastavení polygon módu (VK_POLYGON_MODE_FILL), typu cullingu (VK_CULL_MODE_BACK_BIT) a také pořadí vrcholů pro vykreslování (VK_FRONT_FACE_COUNTER_CLOCKWISE).

5.4.6 Fragment shader

Fragment shader je aplikován na všechny fragmenty, které po předchozí operaci stále zůstaly k použití. Fragment shader také rozhodne o tom, co zapíše do framebufferu a také zda se bude jednat o barvu, nebo o hloubkovou informaci. K tomu využije data z vertex shaderu, které navíc mohou být obohaceny o data jako normály pro světla nebo souřadnice textur.

5.4.7 Barevné prolnutí

Zde dochází k poslední operaci v grafické pipeline. Jedná se o operace se všemi fragmenty na daném pixelu. Tyto fragmenty je možné mixovat mezi sebou, použít průhlednost, znásobit je

nebo přepisovat jeden fragment druhým. U této operace asistují dvě struktury VkPipelineColorBlendAttachmentState a VkPipelineColorBlendStateCreateInfo.

5.4.8 Dokončení grafické pipeline

Co se týče shaderů, je potřeba v grafické pipeline vyplnit ještě jednu strukturu. Jedná se o VkPipelineShaderStageCreateInfo jež určuje parametry použitých shaderů. V mém případě používám v aplikaci dva shadery – vertex a fragment shader, proto je tato struktura opět použita jako pole o dvou elementech. Pro každý shader je potřeba specifikovat následující informace:

- sType – jasně daný typ struktury pro shader
- stage – určuje, do jaké části grafické pipeline je shader zakomponován
- module – specifikuje modul, který obsahuje kód shaderu
- pName - název funkce, jež je nutné zavolat pro fungování shaderu (obvykle „main“)

Jakmile dokončíme iniciaci jednotlivých shaderů, je možné přejit přímo k finálnímu vytvoření grafické pipeline. To provedeme funkcí vkCreateGraphicsPipelines, která přebírá celkem šest parametrů. Jedná se o logickou jednotku, strukturu VkPipelineCacheCreateInfo, createInfoCount, strukturu VkGraphicsPipelineCreateInfo, nullpointer (jelikož pátý parametr nepoužíváme) a samotný objekt grafické pipeline.

5.5 Zpracování a vykreslení scény

Po vytvoření grafické pipeline se dostáváme do poslední kapitoly Vulkan API v mé práci. V této části se sloučí všechny předchozí kapitoly dohromady a dojde k vykreslení požadovaného obrazu modelu a jeho uložení do OpenCV cv::mat matice. Je možné také využít obraz hloubkové mapy. Veškerou práci, kterou Vulkan API v této části odvádí, je opět nutné nahrát pomocí command bufferu. Jako první tedy dojde k vytvoření command bufferu, alokaci parametrů pro command buffer a následné spuštění nahrávání příkazů.

Poté přichází na řadu vytvoření a nastavení struktury VkRenderPassBeginInfo potřebné pro spuštění renderpassu, který byl vytvořen v předcházející kapitole. V této struktuře je nutné předat pointer na objekt renderpassu a objekt framebufferu a dále nastavit výslednou šířku a výšku obrazu, popřípadě hodnotu barvy pixelů, s nimiž se nijak nepracuje (jsou prázdné). Funkcí vkCmdBeginRenderPass spustíme renderpass. Tato funkce má celkem tři parametry. Jedná se o objekt command bufferu, pointer na vytvořenou strukturu VkRenderPassBeginInfo a specifikaci, jakým způsobem budou nahrány příkazy command buffer při prvním průchodu renderpassem.

5.5.1 Nastavení viewport a nůžkového testu

Ve Vulkan API existují další dvě proměnné, které ovlivňují co a jakým způsobem bude vykresleno. Jedná se o VkViewpoer a VkRect2D použitý jako nůžkový test (z anglického scissor test).

Obrázek 5: Ukázka nastavení různého viewport a scissor testu.

Viewport slouží k tomu, abychom mohli nastavit do jaké části framebufferu budeme zapisovat výsledná data. Obecně zapisujeme do celého framebufferu a to je i případ mé aplikace. Proto je velikost viewport okna nastavena od souřadnic 0, 0 (levý horní roh) až k nastavené šířce a výšce (pravý dolní roh), což dohromady vytvoří plochu celého framebufferu. Hodnoty minDepth a maxDepth slouží pro nastavení rozsahu hloubkových dat ve výsledném obraze. Zajímavostí je, že maxDepth může být menší než minDepth a v případě tohoto nastavení způsobí výpočet hloubkových dat obráceně.

```
VkViewport viewport = {};
viewport.height = (float)height;
viewport.width = (float)width;
viewport.minDepth = (float)0.0f;
viewport.maxDepth = (float)1.0f;
vkCmdSetViewport(commandBuffer, 0, 1, &viewport);
```

Výpis 22: Ukázka inicializace VkViewport.

Nůžkový test je oproti OpenGL ve Vulkan API povinný a nelze jej vypnout. Tento test omezí generování fragmentů pomocí fragment shaderů pouze na jasně vymezenou plochu. Vše ostatní mimo tuto plochu je zahozeno a dál se nepoužije. Pokud neexistuje speciální využití v aplikaci, nastavuje se scissor test obecně na stejné hodnoty, jako VkViewport.

```
VkRect2D scissor = {};
scissor.extent.width = width;
scissor.extent.height = height;
vkCmdSetScissor(commandBuffer, 0, 1, &scissor);
```

Výpis 23: Ukázka inicializace nůžkového testu.

5.5.2 Příprava k vykreslení

Před námi je pouze pár posledních příkazů, jež jsou nutné pro finální vykreslení obrazu. Po všech konfiguračních krocích už stačí jen „připnout“ jednotlivé funkce do command bufferu a odeslat je ke zpracování. Jedná se o:

- vkCmdBindPipeline
- vkCmdBindVertexBuffer
- vkCmdBindIndexBuffer
- vkCmdBindDescriptorSets
- nastavení MVP matice a světla
- vykreslení každé části příslušného modelu
- ukončení renderpassu + command bufferu a odeslání ke zpracování

Funkce vkCmdBindPipeline pošle do command bufferu požadavek na spuštění celé grafické pipeline. Dojde k průchodu všech nastavení pipeline, „aktivaci“ všech vytvořených shaderů a nastavení všech potřebných testů či operací. Jedná se například o nastavení prolnutí, nebo již výše zmíněného nůžkového testu.

Funkce vkCmdBindVertexBuffer pošle do command bufferu požadavek na spuštění nastavení celého vertex (vrcholového) bufferu. V této chvíli dojde k navázání celého vertex bufferu a jeho atributů na správné descriptory.

Funkce vkCmdBindIndexBuffer podobně jako předchozí funkce, pošle do command bufferu požadavek na spuštění nastavení index (indexového) bufferu.

Funkce vkCmdBindDescriptorSets pošle do command bufferu požadavek na spuštění všech vytvořených descriptor setů a jejich descriptorů.

Nyní přichází na řadu nastavení souřadnic pro model, pohled kamery a projekci. Nastavení může vypadat následovně:

```
glm::mat4 model = glm::mat4(1.0);
glm::mat4 view = glm::lookAt(lookAtVector, glm::vec3(0.0f, 1.0f, 0.0f), glm::
    vec3(0.0f, 1.0f, 0.0f));
glm::mat4 projection = glm::perspective(glm::radians(45.0f), (float)width / (
    float)height, 0.1f, 256.0f);
projection[1][1] *= -1;
```

Výpis 24: Ukázka nastavení MVP matice.

Zde považuji za důležité zmínit poslední řádek kódu. Knihovna GLM, kterou v této práci používám, byla původně napsána pro OpenGL, kde jsou Y souřadnice prohozeny. Ve Vulkan API je tedy potřeba přehodit Y souřadnici zpět, aby výsledný vykreslený model nebyl vzhůru nohama.

Po nastavení MVP matice v rámci GLM knihovny předáme tyto informace do objektu uniform bufferu, se kterým aplikace pracuje, viz.

```
ubo.lightPosition = glm::vec3(3, 3, 1);
ubo.model = model;
ubo.view = view;
ubo.proj = projection;
```

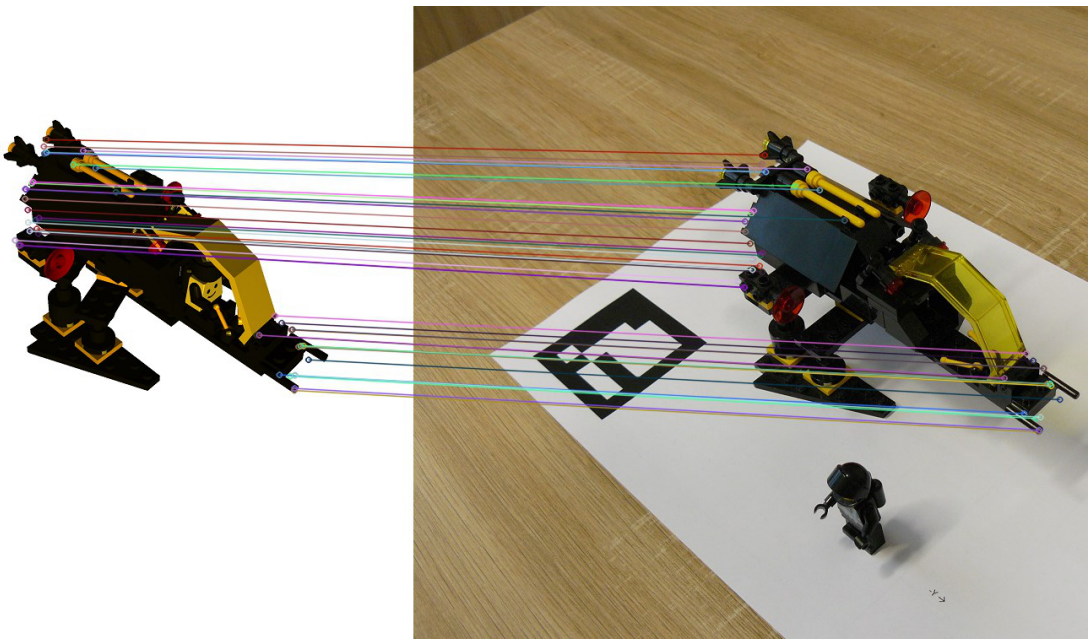
Výpis 25: Ukázka nastavení pozice světla a předání MVP matice do objektu uniform bufferu.

Jako poslední se provede programová smyčka, která pro každou část načteného modelu provede dvě funkce. Nejprve pomocí funkce vkCmdPushConstants pošleme do command bufferu informaci o tom, jakou texturu má pro aktuální část modelu využít fragment shader. Následně je pomocí funkce vkCmdDrawIndexed vykreslena konkrétní část 3D modelu se svou případnou texturou. Ve funkci vkCmdDrawIndexed předávám informaci o jaký command buffer pro nahrání příkazu se jedná, dále počet indexů, který má být vykreslen a první index od kterého má vykreslování začít.

Po kompletním vykreslení nejprve ukončíme renderpass, následně nahrávání command bufferu a poté odešleme příkazy v command bufferu ke zpracování.

6 Obrazové detektory a deskriptory

V této části se zabývám využitím několika různých detektorů a deskriptorů zobrazovaných objektů pro následné vyhledávání shod ve dvou různých obrazech. Tato kapitola je rozdělena na dvě části. První část pojednává o algoritmech dostupných v knihovně OpenCV. Druhá část představuje dva nové algoritmy z posledních tří let. Jeden algoritmus slouží pro detekci a druhý pro deskripci klíčových bodů.



Obrázek 6: Detekce a napárování společných klíčových bodů pomocí algoritmu AKAZE.

6.1 Detektory a deskriptory dostupné v OpenCV

Díky popularitě OpenCV a velikosti komunity se do této knihovny již dostalo velké množství různých detektorů či deskriptorů klíčových bodů. Jedná se konkrétně o detektory a deskriptory ORB, KAZE, AKAZE, SIFT, SURF, FAST, BRISK, FREAK, GFTT, SimpleBlob, MSER a STAR. SIFT a SURF jsou patentované algoritmy, které lze použít bezplatně pouze pro účely výzkumu a akademické práce. Tyto algoritmy byly odstraněny ze základní verze OpenCV a nyní je lze nalézt v přidruženém repozitáři ve službe Github se jménem „Opencv_contrib“. V závěrečné tabulce bude porovnána efektivita a čas nutný k dosažení výsledků u všech detektorů či deskriptorů. Jednotlivé detektory a deskriptory z knihovny OpenCV jen velmi stručně představím, jelikož podrobný popis fungování těchto detektorů a deskriptorů by sám o sobě rozsahem zabral enormní množství textu, což není cílem této pasáže. Pro zájemce přidávám odkazy na implementace jednotlivých detektorů či deskriptorů ORB[17], KAZE[25], AKAZE[18], SIFT[19], SURF[20], FAST[21], BRISK[26], FREAK[27], GFTT[28], SimpleBlob[29], MSER[30] a STAR[31].

6.1.1 ORB

ORB neboli Oriented FAST and Rotated BRIEF je binární algoritmus. Jedná se o spojení detektoru klíčových bodů metody FAST[21] a deskriptorů metody BRIEF[22] s několika úpravami k dosažení většího výkonu. [23]. Autory tohoto algoritmu jsou Ethan Rublee, Vincent Rabaud, Kurt Konolige a Gary R. Bradski. ORB algoritmus nejprve použije metodu FAST k nalezení klíčových bodů, následně aplikuje Harrisův rohový detektor[24] k nalezení těch nejlepších klíčových bodů. Poté je využito pyramidové schéma k nalezení multiscale vlastností. U metody FAST je problém v tom, že nevypočítává orientaci bodů. ORB algoritmus tento problém řeší pomocí výpočtu vážené intenzity centroidu s rohovým bodem lokalizovaným v centru testované oblasti. Díky tomu získáme směrový vektor mířící z rohového bodu k centroidu, který nám poskytne danou orientaci bodu. Jak již bylo psáno, ORB využívá dekriptory metody BRIEF. Nicméně i zde došlo k vylepšení, jelikož metoda BRIEF nepodává sama o sobě dobré výsledky při rotacích. ORB algoritmus toto chování vylepšuje tím, že řídí BRIEF metodu dle orientace klíčových bodů. Pro jakýkoliv set vlastností o N binárních testech v nějakém konkrétním místě (např.: x_i, y_i), definuje $2 \times N$ matici M, která obsahuje souřadnice pixelů. Poté je pomocí orientace v proměnné θ nalezena příslušná matice rotace a díky této matici je natočena také matice M.

6.1.2 KAZE

KAZE je algoritmus, který je schopný použít detektor i deskriptor klíčových bodů. Jedná se o binární algoritmus a využívá nelineární difuze k detekci a popisu vlastností daného obrazu v nelineárním prostoru. Umí se také vypořádat se šumem v obraze. Autory algoritmu jsou Pablo F. Alcantarilla, Jesús Nuevo a Adrien Bartoli.

6.1.3 AKAZE

Akaze je zkratka pro Accelerated KAZE algoritmus. Je to v podstatě vylepšený KAZE algoritmus. Opět je binárním algoritmem. KAZE zcela funguje v nelineárním měřítku. Prostřednictvím nelinerání difůze detekuje a popisuje požadované obrazové vlastnosti. AKAZE využívá novější typ výpočtu, tzv. Fast Explicit Diffusion (FED), který taktéž využívá pyramidové schéma, což dramaticky urychluje výsledný výpočet. V algoritmu také dochází k výpočtu tzv. Modified-Local Difference Binary(M-LDB) deskriptoru, který využívá informace o gradientu z výpočtu provedeného FED. AKAZE dosahuje srovnatelných výsledků, jako původní KAZE algoritmus, ale jeho výpočet je několikanásobně rychlejší. [25]

6.1.4 SIFT

SIFT je označení pro Scale Invariant Feature Transform algoritmus, který vytvořil D. Lowe v roce 1999. Algoritmus obsahuje čtyři základní kroky. Jedná se o detekci extrémního měřítka, nalezení klíčových bodů, přiřazení orientace bodů a vytvoření deskriptoru klíčových bodů. Pro

nalezení extrémního měřítka využívá algoritmus diferencí gausiánů. Ten získáme pomocí aplikování Gaussova filtru se dvěma různými nastaveními. Pro detekci klíčových bodů je využita Hessianova matice. Všechny nalezené klíčové body jsou prohnány skrz sérii testů a pokud jsou nižší než určitá prahová hodnota, pak jsou zrušeny. Díky tomu zůstanou ke zkoumání pouze „kvalitní“ klíčové body. Pro přiřazení orientace jednotlivým bodům je vypočtena magnituda gradientu a také směr bodu. Jako deskriptor je použita matice o 16×16 bodech okolo zkoumaného klíčového bodu. Tato matice je následně rozdělena na menší matice o velikosti 4×4. Pro každou matici je vytvořen histogram. Mimo tento výpočet dochází k několika dalším opatřením, aby byl výpočet robustnější a odolný například proti změnám světla či rotaci obrazu.

6.1.5 SURF

SURF je zkratkou pro Speed-Up Robust Features. SURF je v podstatě zrychlenou verzí SIFT algoritmu. Tento algoritmus vznikl v roce 2006 (SURF: Speeded Up Robust Features) a jeho autory jsou Herbert Bay, Tinne Tuytelaars and Luc Van Gool.

SURF pro aproximaci Gaussovy křivky využívá box filtr. Výhodou této aproximace je, že konvoluce s box filtrem může být jednoduše vypočítána s pomocí integrálních obrazů. Toto navíc může být prováděno paralelně pro různé velikosti filtru. SURF taktéž používá Hessianovu matici. Pro zjištění orientace jednotlivých bodů používá SURF vlnové odrazy ve vodorovném i svislém směru s okolím o velikosti 6 dalších bodů. Pro deskriptory pak SURF používá opět vlnové odrazy ve vodorovném i svislém směru. Rozdíl je v tomto případě velikost měřeného okolí, které je u deskriptoru až 20 dalších bodů po obou osách kolem klíčového bodu. Následně je tato matice rozdělena do menších matic po 4×4 bodech, ze kterých jsou vytvořeny vektory pro další zpracování. Díky tomuto přístupu má SURF deskriptor 64 dimenzí. Čím méně dimenzí bude, tím rychlejší bude také algoritmus. Nicméně je to za cenu horšího výsledku.

6.1.6 FAST

FAST znamená „Features from Accelerated Segment Test“ a funguje na principu rychlého vyhledávání rohových hran. Autory tohoto algoritmu jsou Edward Rosten a Tom Drummond. Algoritmus pracuje s kruhovou oblastí o velikosti 16 pixelů kolem hlavního pixelu, který označuje za rohový a využívá k tomu čtyři parametry.

- threshold - rozdílový práh mezi intenzitou hlavního pixelu oproti kruhu kolem něj
- nonMaxSuppression - true nebo false hodnota. pokud je true, pak je aplikována na detekované rohy
- type - jeden ze tří typů sousedství okolo hlavního pixelu

6.1.7 BRISK

BRISK je binární detektor i deskriptor klíčových bodů. Celým názvem se jedná o „Binary Robust Invariant Scalable Keypoints“. BRISK funguje na základě ručně vytvořené kruhové vzorkovací sítě. Pro každý pixel, který chceme prověřit, je vybrána menší oblast kolem tohoto pixelu, která je následně rozostřena pomocí Gaussova filtru. Čím dále jsou jednotlivé pixely od hlavního porovnávaného pixelu, tím větší je potřeba použít Gaussův filtr na rozostření. V rámci použití této kruhové sítě BRISK rozlišuje blízké a vzdálené páry pixelů. Vzdálené páry používá k výpočtu orientace a blízké páry používá k porovnání intenzity, což je důležité pro deskriptor.

6.1.8 FREAK

FREAK neboli „Fast Retina Keypoint Descriptor“ je pouze deskriptor. Sám nedokáže detekovat žádné klíčové body. Tento algoritmus je hodně podobný BRISK i ORB algoritmům. Z BRISK si přebírá vzory pro vzorkování a také mechanismus orientace. Z ORB si bere techniku strojového učení pro naučení optimálních vzorkovacích párů. FREAK pracuje se vzorkováním, které je podobné tvarem sítnici lidského oka. Jednotlivé kruhové vzorky jsou mnohem početnější ve středu sítnicové sítě a čím blíže jsou testované vzorky k okraji, tím méně se jich tam také nachází.

6.1.9 GFTT

GFTT je zkratka pro „Good features to track“. Tento algoritmus se zabývá nalezením významných rohů nacházejících se v testovaném obraze. Algoritmus si spočítá u každého nalezeného rohu svou kvalitu. Následně jednotlivé nalezené rohy prochází a třídí dle kvality. Při třídění dochází k zahození všech rohů, které mají menší kvalitu než jiný silnější roh v blízké vzdálenosti. Tato vzdálenost se dá ovlivnit nastavení parametru „maxDistance“.

6.1.10 SimpleBlob

SimpleBlob je starší a jednodušší algoritmus zaměřující se na vyhledávání shluků dat v obrazech. Ze všech testovaných detektorů či deskriptorů je tento možný nastavit na nejvíce způsobů. Obsahuje celkem 19 parametrů, jimiž je možné výsledky ovlivnit. Je možné nastavit například vyhledávání podle minimální či maximální oblasti, vyhledávání podle barvy, minimálního a maximálního prahování nebo vzdálenosti jednotlivých shluků od sebe.

6.1.11 MSER

MSER značí zkratku pro „Maximally stable extremal region extractor“. MSER algoritmus existuje ve dvou řešeních. Jedno pro obrazy v šedých barvách a druhé pro klasické barevné obrazy. Algoritmus obsahuje několik parametrů pro nastavení kontrolovaných oblastí, jejich nejmenší a největší velikost, variaci velikosti, diverzitu barevnosti pixelů nebo například nastavení velikosti

rozostření hran. Zajímavostí je, že tento algoritmus vznikl v Praze na ČVUT. Autory jsou J. Matas, O. Chum, M. Urban a T. Pajdla. [30]

6.1.12 STAR

STAR je pouze detektor klíčových bodů. Tento algoritmus je stejně jako ostatní implementován v OpenCV a je možné mu nastavit několik parametrů pro upřesnění či optimalizaci detekce. Jedná se například o maximální velikost detektoru v pixelech či nastavení prahu detektoru.

6.2 ULATCH deskriptor a TILDE detektor

V této menší kapitole představím dva nové algoritmy. Jedná se o deskriptor klíčových bodů ULATCH a detektor klíčových bodů TILDE. Oba algoritmy jsou založeny na strojovém učení. Tyto algoritmy vznikly v posledních třech letech a jsou volně dostupné přímo od jejich autorů, nebo mají své implementace v různých programovacích jazycích nasdílené na komunitní službě GitHub.

6.2.1 ULATCH

V této části se dostáváme k prvnímu algoritmu, jež není součástí OpenCV. ULATCH [1] je zkratka pro „Upright learned Arrangements of Three Patch Codes“ a je to vylepšená verze LATCH algoritmu z roku 2015. LATCH i ULATCH jsou pouze deskriptory. Není s nimi tedy možné detekovat klíčové body. Pro tento případ je nutné využít zaběhnutého algoritmu typu ORB či SIFT. V mé práci jsem otestoval obě kombinace a výsledky popisuji níže v kapitole výsledků. ULATCH funguje na základě tzv. trojic. Klasické binární deskriptory jsou velmi závislé na šumu nebo menších lokálních barevných změnách. Stačí změna jediného pixelu a výsledný deskriptor bude vypadat zcela jinak. Tento problém se snaží řešit ULATCH tím, že neporovnává jednotlivé pixely, ale v oblasti, kterou zpracovává, si vybírá několik trojic pixelů, což jsou 3x3 matice a s těmi následně dále pracuje. Ulatch je vysoce optimalizovaný multivláknový algoritmus využívající instrukce typu SSE2/3/4/4.1, AVX, AVX2. Díky tomuto faktu je možné jej spustit pouze na procesorech, které těmito instrukcemi disponují.

6.2.2 TILDE

TILDE [2] je zkratka pro Temporally Invariant Learned DEtector. Jedná se o druhý algoritmus mimo OpenCV a také o novější algoritmus z roku 2015. Jeho autory jsou Yannick Verdie, Kwang Moo Yi, Pascal Fua a Vincent Lepetit. Algoritmus je založen na strojovém učení a primárně slouží pro detekci klíčových bodů, které se v obraze objevují i při velkých změnách počasí či množství dopadajícího světla s čímž má většina dnešních detektorů stále velký problém. Tvýrci tohoto algoritmu veřejně poskytli i datasety, na kterých je algoritmus trénovaný a lze díky tomu detekovat i objekty v mé práci.

7 Výsledky testů jednotlivých detektorů a deskriptorů

V této sekci porovnáme výsledky jednotlivých algoritmů. Vznikly dvě tabulky. První tabulka zobrazuje rychlost jednotlivých algoritmů při hledání a párování klíčových bodů. Druhá tabulka zobrazuje efektivitu napárovaných klíčových bodů a správně napárovaných klíčových bodů. Tabulky reflektují také to, že některé algoritmy slouží pouze jako detektory a některé naopak pouze jako deskriptory. V případě, že je použita kombinace detektoru a deskriptoru, bude to v tabulce vyznačeno.

Metodika měření

Metodika měření efektivity jednotlivých algoritmů se v mé práci skládá z několika kroků. Nejprve jsem se snažil pro náš konkrétní test dvou obrazů najít optimální nastavení jednotlivých detektorů či deskriptorů pouze na základě jejich parametrů. To bylo provedeno pomocí zanořených for cyklů, které postupně po vhodně zvolených krocích procházely a zvyšovaly nastavení jednotlivých parametrů. U některých algoritmů (například KAZE) bylo vzhledem k jejich extrémní časové náročnosti výpočtu přistoupeno k testu dle náhodně zvolených parametrů, na základě kterých následně došlo k upřesnění parametrů, ve kterých algoritmus podává dobré výkony. Díky těmto krokům jsem se dostal k dobrým výsledkům, které dokáže daný algoritmus poskytnout pouze na základě jeho parametrů. Pro určení rychlosti detekce a deskripce jsem použil časovač z knihovny chrono z jazyka C++. Vysledná rychlost detekce i deskripce v první tabulce byla měřena pro oba testované obrázky najednou. Následně byla pro potřeby této práce vytvořena druhá kontrola. Tato kontrola slouží k tomu, aby porovnala souřadnice nalezeného klíčového pixelu na ose x a y v obou obrazech. Pokud se jedna či druhá souřadnice liší o více jak 10 bodů, pak je takový klíčový bod vyřazen. Pokud se klíčový bod do nastaveného rozsahu vejde, pak je vložen do nového vektoru správných klíčových bodů. Vektor s těmito správnými klíčovými body je následně použit pro vyobrazení finálních shod klíčových bodů mezi oběma obrazy. Všechny výsledky byly změřeny na obrázcích o rozměrech 1076×807 pixelů a stolním počítači s konfigurací Intel Core i5-4690K, nVidia GTX 970 a 16 GB RAM.

Tabulka 1: Rychlost detekce a deskripce jednotlivých detektorů a deskriptorů.

Algoritmus	Rychlost detekce obou obrazů dohromady [ms]	Rychlost deskripce obou obrazů dohromady [ms]
KAZE	13035	2163
AKAZE	582	1313
SIFT	1713	1.3
SURF	388	209
ORB	306	1.1
BRISK	65	31
FAST + ORB	18	42
ORB + FREAK	464	116
GFTT + ORB	1581	34
MSER + ORB	747	88
SimpleBlob + ORB	8970	44
STAR + ORB	165	37
ORB + BoostDesc	352	1146
SIFT + BoostDesc	1122	1090
ORB + ULATCH	562	274
SIFT + ULATCH	969	61
TILDE + ORB	3978	36
TILDE + SIFT	3978	571

Tabulka 2: Efektivita jednotlivých detektorů a deskriptorů.

Algoritmus	Počet nalezených klíčových bodů v prvním a druhém obraze	Počet napárovaných klíčových bodů	Počet správně napárovaných klíčových bodů
KAZE	1042 & 811	512	187
AKAZE	724 & 899	479	130
SIFT	200 & 200	96	17
SURF	412 & 702	285	68
ORB	733 & 750	327	43
BRISK	384 & 516	181	63
FAST + ORB	493 & 503	206	40
ORB + FREAK	424 & 415	204	46
GFTT + ORB	104 & 101	51	7
MSER + ORB	580 & 1710	190	18
SimpleBlob + ORB	10 & 946	9	1
STAR + ORB	797 & 740	333	52
ORB + BoostDesc	202 & 202	70	16
SIFT + BoostDesc	200 & 200	111	7
ORB + ULATCH	1923 & 1951	1923	18
SIFT + ULATCH	365 & 500	365	4
TILDE + ORB	388 & 494	111	27
TILDE + SIFT	500 & 500	258	96

Tabulka 3: Nastavení parametrů jednotlivých detektorů a deskriptorů.

Algoritmus	Parametry
KAZE	kaze = KAZE::create(true, true, 0.001, 4, 4);
AKAZE	akaze = AKAZE::create();
SIFT	sift = xfeatures2d::SIFT::create(200, 3, 0.02, 5, 1.6);
SURF	surf = xfeatures2d::SURF::create(500, 6, 2, 1, 1);
ORB	orb = ORB(750, 1.82403, 3, 77, 0, 3, HARRIS_SCORE, 48, 20);
BRISK	brisk = BRISK::create(40, 0, 8);
FAST + ORB	fast = FastFeatureDetector::create(93, 0, 2); orb = ORB::create();
ORB + FREAK	orb = ORB::create(); freak = xfeatures2d::FREAK::create();
GFTT + ORB	GFTT = GFTTDetector::create(1000, 0.1, 1, 15, true, 0.1) orb = ORB::create();
MSER + ORB	mser = MSER::create(2, 5, 2350, 0.2); orb = ORB::create();
SimpleBlob + ORB	simpleBlob = SimpleBlobDetector::create(params) orb = ORB(300, 1.2f, 8, 200, 0, 3, ORB::HARRIS_SCORE, 300, 20);
STAR + ORB	star = xfeatures2d::StarDetector::create(20, 27, 77, 10, 2); orb = ORB::create();
ORB + BoostDesc	orb = ORB(300, 1.2f, 8, 200, 0, 3, HARRIS_SCORE, 300, 20); boost = BoostDesc(xfeatures2d::BoostDesc::BGM, true, 0.75f);
SIFT + BoostDesc	sift = xfeatures2d::SIFT::create(200, 3, 0.04, 10, 1.6); boost = BoostDesc(xfeatures2d::BoostDesc::BGM, true, 4.75f);
ORB + ULATCH	ORB(5000, 1.2, 8, 80, 0, 2, cv::ORB::HARRIS_SCORE, 80, 20); ULATCH výchozí nastavení, multithreading = ano
SIFT + ULATCH	sift = cv::xfeatures2d::SIFT::create(500, 3, 0.02, 5, 1.6 ULATCH výchozí nastavení, multithreading = ano
TILDE + ORB	StLouis24 dataset orb = ORB(300, 1.27, 1, 100, 0, 4, HARRIS_SCORE, 199, 20);
TILDE + SIFT	StLouis24 dataset sift = xfeatures2d::SIFT::create(200, 3, 0.02, 5, 1.6);

Vliv šumu na výsledky párování

V rámci dalšího testování detektorů či deskriptorů jsem vyzkoušel vliv šumu v obraze založeném na normálním rozdělení a jeho následné porovnání s vygenerovaným obrazem z Vulkan aplikace. Pro provedení testu bylo nejprve potřeba z vygenerovaného obrazu vytvořit nový zašumělý obrázek dle následujícího zdrojového kódu.

```
std::random_device rd{};
std::mt19937 gen{ rd() };
double stdd = 10.0;
Vec<unsigned char, 3> mean;
for (int i = 0; i < template_image.rows; ++i){
for (int j = 0; j < template_image.cols; ++j){
mean = template_image.at<Vec3b>(i, j);
std::normal_distribution<> d0{ static_cast<double>(mean[0]), stdd };
std::normal_distribution<> d1{ static_cast<double>(mean[1]), stdd };
std::normal_distribution<> d2{ static_cast<double>(mean[2]), stdd };

noisyMat.at<Vec3b>(i,j) = Vec3b(saturate_cast<uchar>(d0(gen)),
saturate_cast<uchar>(d1(gen)), saturate_cast<uchar>(d2(gen)));
   }
}
```

Výpis 26: Ukázka vygenerování zašumělého obrazu.

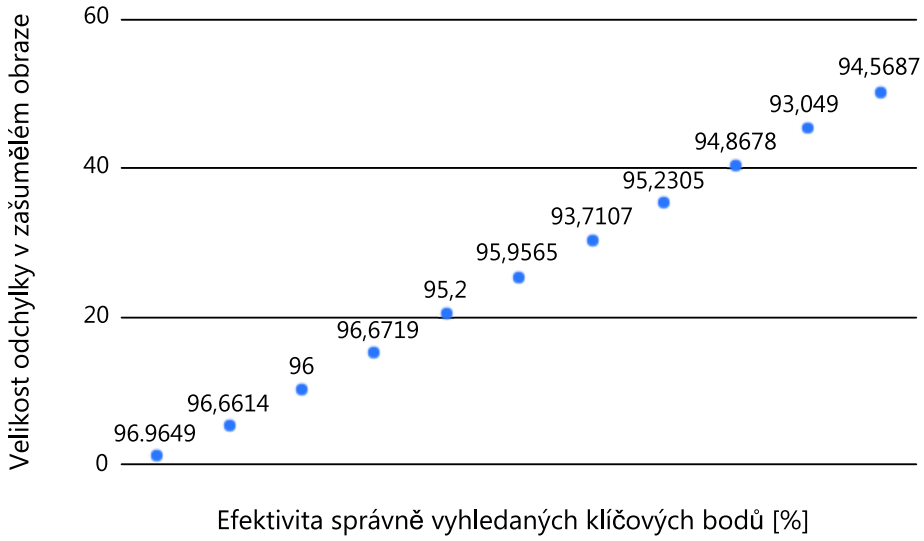
Následně byly oba obrázky podrobeny testu algoritmů AKAZE, SURF a TILDE. V případě, že bychom porovnali dva identické obrázky, dostali bychom 100% shodu v nalezených bodech. Zanesený šum do jednoho z obrázků ukázal, že na algoritmy skutečně vliv má, i když není moc velký. Maximální zhoršení bylo zhruba 3,7%, přičemž nejlépe dopadl algoritmus SURF, jenž se zhoršil pouze o 2,7%. Tento 1% nárůst lze přičíst tomu, že algoritmus SURF jako jediný z výše jmenovaných pracuje s obrázky pouze v šedých odstínech a díky tomu mu zašumění obrazu tolik nevadí. Zajímavé je také pozorovat výsledek algoritmu TILDE, jenž jako jediný nalezl v zašumělém obrazek několik desítek klíčových bodů, které s ničím nesouvísí a není je možné na nic napárovat. U algoritmu AKAZE bylo nalezeno celkem 606 správných klíčových bodů. U algoritmu SURF to bylo 353 správných klíčových bodů a u algoritmu TILDE se jedná o 318 správných klíčových bodů.

Obrázek 7: Ukázka výsledku algoritmu SURF se zašumělým obrázkem.

Obrázek 8: Ukázka výsledku algoritmu AKAZE se zašumělým obrázkem.

Obrázek 9: Ukázka výsledku algoritmu TILDE se zašumělým obrázkem.

K testu se zašumělým obrazem byl zhotoven ještě jeden test. Bylo vytvořeno celkem 10 různých zašumělých obrazů, z nichž každý měl jiné nastavení standardní odcholky pro generovaný šum. Standardní odchylka začínala na hodnotě 1 a skončila na hodnotě 50, přičemž každý krok zvedl odchylku o 5 bodů. Pro každý vygenerovaný obraz byl následně spuštěn algoritmus AKAZE, který porovnal vygenerovaný obraz z Vulkan API s vygenerovaným zašumělým obrazem. Výsledky byly zaznamenány do grafu níže.



Obrázek 10: Graf závislosti standardní odchylky v zašumělém obraze na počet správně napárovaných bodů.

8 Závěr

Na výsledky v předcházející kapitole se lze dívat několika způsoby. První způsob je sledovat počet bodů, které byly nakonec správně napárovány. V tomto případě by vyhrály algoritmy KAZE, AKAZE a TILDE se SIFT deskriptorem. Nicméně tento pohled je omezený pouze na tuto diplomovou práci a nevypovídá tolik o možnostech použití v reálných proměnlivých podmínkách (například jiná velikost modelu versus reálný objekt, jiná pozice v každém obraze atd.). Pokud se na výsledky podíváme bez závěrečného testu kontroly souřadnic jednotlivých klíčových bodů v každém obraze a naopak se pouze vizuálně podíváme na výsledky, které tyto algoritmy dokáží dle nastavených podmínek zobrazit, pak se nejlépe umístí AKAZE a KAZE. Tyto dva algoritmy dokázaly při mém testování zobrazit nejvíce napárovaných a zároveň relevantních klíčových bodů, aniž by bylo potřeba kontrolovat souřadnice klíčových bodů. Ani tento způsob ale nepovažuji za kompletně správný. Ke všem výsledkům testů je dle mého názoru potřeba přičíst ještě jednu komponentu, jíž je čas. Pokud chceme provádět smysluplné porovnávání obrazu (například z mnoha různě nasnímaných pohledů kamery najednou) vůči vygenerovaném modelu, pak je potřeba získat nejen hodně relevantních napárovaných klíčových bodů, ale také je potřeba tuto operaci provádět co nejrychleji. Pokud jako poslední přičtu k výsledkům čas, který algoritmy potřebují pro zpracování vstupních dat, vychází vítězně algoritmy AKAZE, SURF nebo kombinace STAR + ORB a kombinace TILDE + SIFT. Algoritmus AKAZE rozhodně není nejrychlejší, ale celková doba zpracování dat okolo dvou sekund společně se 130 správnými páry klíčových bodů je velmi dobrá. Vůbec největší počet správných párů klíčových bodů nalezl algoritmus KAZE. Bohužel díky rychlosti tohoto algoritmu jej lze v praxi nasadit pouze tam, kde nejde o zpracování v reálném čase.

Nejrychlejším detektorem v této práci je dle výše uvedených měření algoritmus FAST s dobou 18 milisekund. Toto zjištění mne překvapilo, jelikož jsem dle konzultací a informací načtených z internetu očekával jako vítěze v kategorii rychlosti detekce algoritmus ORB. Ten se nakonec nachází až na třetím místě za algoritmem BRISK. Nejrychlejším deskriptorem je algoritmus ORB s rychlostí 1,1 milisekundy, který je jen s minimálním rozdílem následovaný algoritmem SIFT s rychlostí 1,3 milisekundy. Poté již vznikají mezi jednotlivými deskriptory značné rozdíly v rychlosti. Nejpomalejším detektorem je zcela jistě původní algoritmus KAZE s časem 13 sekund. Za ním je o 4 sekundy rychlejší SimpleBlob algoritmus. Nejpomalejším deskriptorem je s časem v průměru okolo dvou sekund opět algoritmus KAZE. Za zmínku stojí rychlost deskriptoru Ulatch, jenž běží pouze za pomoci CPU a specifických instrukce typu AVX, AVX2. Tento deskriptor v kombinaci s detektorem SIFT potřeboval pro deskripci klíčových bodů pouze 61 milisekund a překonal tak rychlostně polovinu testovaných algoritmů. Pro účely mé práce označuji za nejméně vhodný algoritmus SimpleBlob. Tento algoritmus je velmi pomalý a získané výsledky jsou s velkým předstihem nejhorší. Jedná se o jediný algoritmus, u kterého se po kontrole souřadnic klíčových bodů podařilo napárovat pouze jeden správný klíčový bod. V tomto případě se nejedná o žádné překvapení, jelikož je tento algoritmus určený pro vyhledávání jednoduchých větších

vzorů, které se na testovaných obrazech téměř nevyskytují.

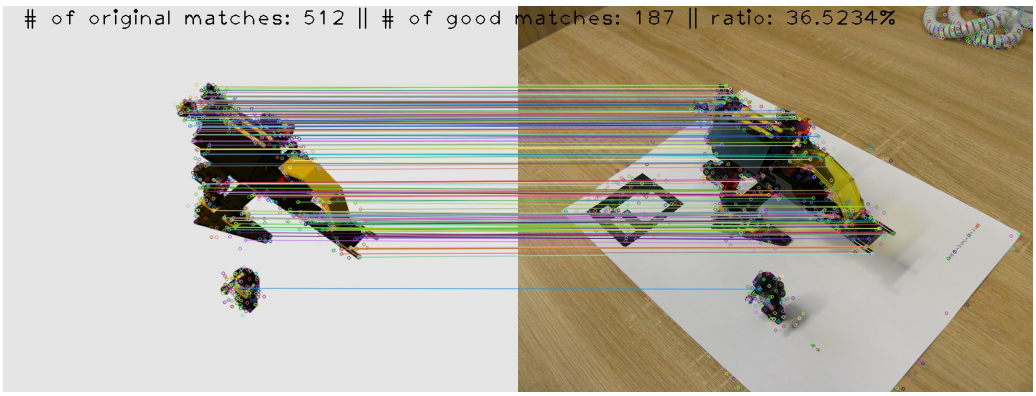
Důležité je zmínit algoritmy ULATCH a TILDE. Oba tyto algoritmy totiž pracuji na bázi strojového učení a strojové učení je pravděpodobně směr, kterým se bude většina nejnovějších detektorů či deskriptorů vydávat. Jedním z prvních velmi schopných strojově učených algoritmů je právě TILDE, který je popsán i vyzkoušen v této práci a nebo také algoritmus LIFT („Learned Invariant Feature Points“), jenž vznikl v roce 2016 [32]. Bohužel implementace algoritmu LIFT je psána pouze v jazyce Python, a proto nebyl pro tuto práci využit.

Cílem této práce bylo také vytvořit aplikaci, která dokáže rychle načítat a zobrazovat 3D modely z různých pohledů kamery pomocí Vulkan API. Takovou aplikaci se mi podařilo vytvořit a otestovat na několika různých modelech. V rámci testování na modelu nanosuit [2] jsem změřil dva časy. Jeden čas pro přegenerování command bufferů a změnu pohledu kamery pro již načtený model v rámci aplikace a druhý čas pro předání nově vygenerovaného pohledu kamery z aplikace do cv::Mat matice pro další zpracování v knihovně OpenCV. Samotné přegenerování pohledu kamery trvá na testovaném modelu zhruba 1,1 milisekundy, což je řádově rychleji než podobná aplikace v OpenGL, která již na půdě Vysoké školy báňské vznikla. Příprava a předání dat z Vulkan API do OpenCV cv::Mat matice pak trvá v průměru 5 milisekund pro obraz o velikosti 640×640 pixelů, 11 milisekund pro obraz o velikosti 1024×1024 pixelů a 38 milisekund pro obraz o velikosti 2048×2048 pixelů. Zmíněná aplikace v OpenGL ve většině případů generuje obrazy o průměrné velikosti 400 pixelů na výšku i na šířku. I při těchto menších rozměrech zabere aplikaci v OpenGL příprava dat a předání do cv::Mat matice v průměru 20 milisekund. Z naměřených výsledků je patrné, že Vulkan API lze uvažovat jako vhodnější volbu pro implementaci aplikace tohoto typu. Oproti OpenGL přináší znatelný rychlostní rozdíl ve zpracování dat a potvrzuje tak předpoklad této diplomové práce.

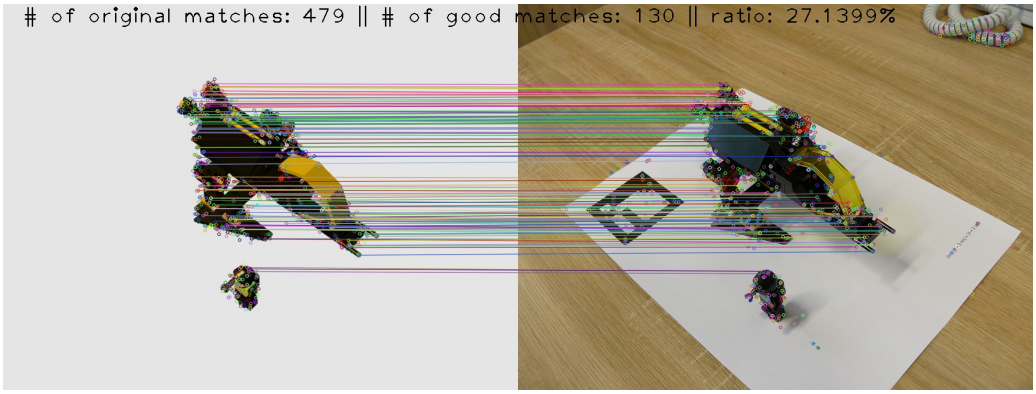
V rámci této diplomové práce jsem se naučil mnoho nových konceptů a osvojil si Vulkan API. Dovolím si tvrdit, že Vulkan API je moderní, rychlé a přímočaré API, které má před sebou velkou budoucnost a lze jej využít na mnoho různorodých grafických či výpočetních úloh. Snaží se řešit spoustu problémů, které se objevily na starých API typu OpenGL (například již zmiňované problémy se shadery) a zároveň je od začátku multiplatformní, což považuji za velkou výhodu. Jako nevýhodu naopak vidím velkou složitost v začátcích a méně početnou komunitu vývojářů, nicméně oba tyto problémy se časem vyřeší. Tato diplomová práce potvrdila předpoklad, že Vulkan API bude pro aplikaci tohoto typu (generování různých pohledů kamery kolem 3D objektu a předávání obrazových dat z framebufferu do cv::Mat matice) vhodnější a rychlejší volbou, což dokazují naměřené výsledky v odstavci výše.

9 Přílohy

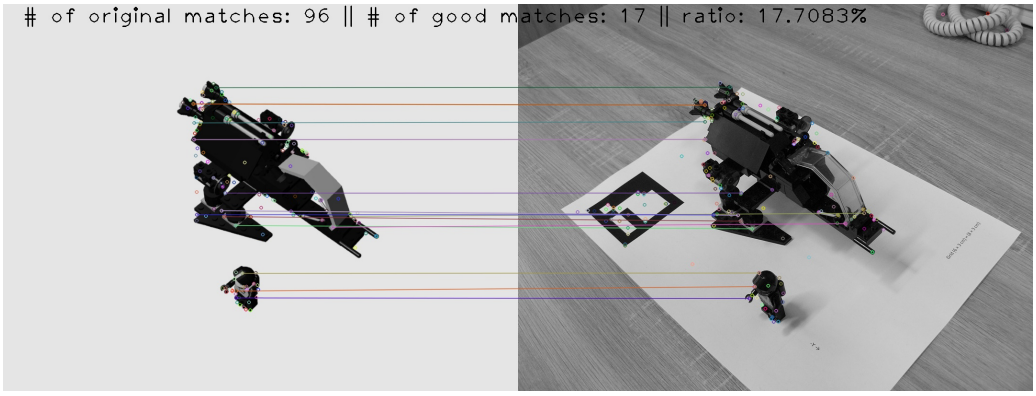
Součástí této práce je CD, na kterém jsou přiloženy zdrojové kódy a použité knihovny. Projekt je možné spustit na počítači s grafickou kartou, která podporuje Vulkan SDK ve verzi 1.0.65 a Cuda SDK ve verzi 9.1. Do příloh také přikládám vizuální výsledky jednotlivých algoritmů.



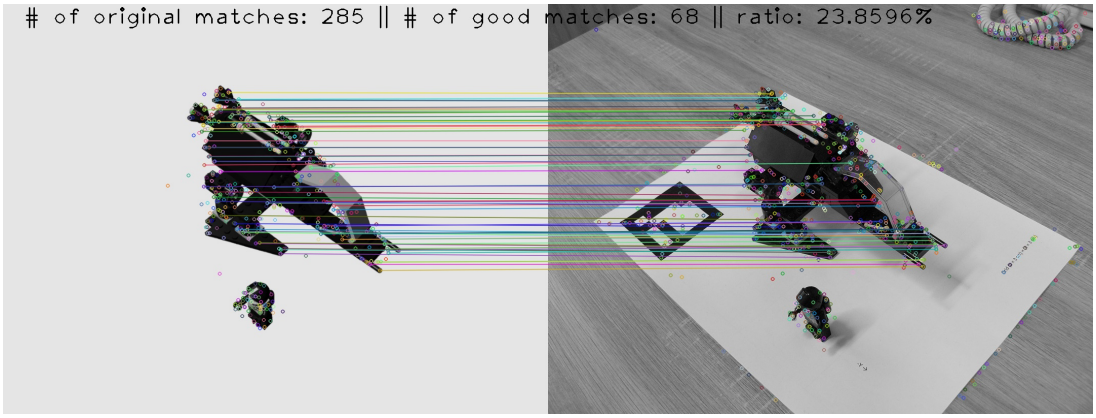
Obrázek 11: Vizuální výsledek algoritmu KAZE.



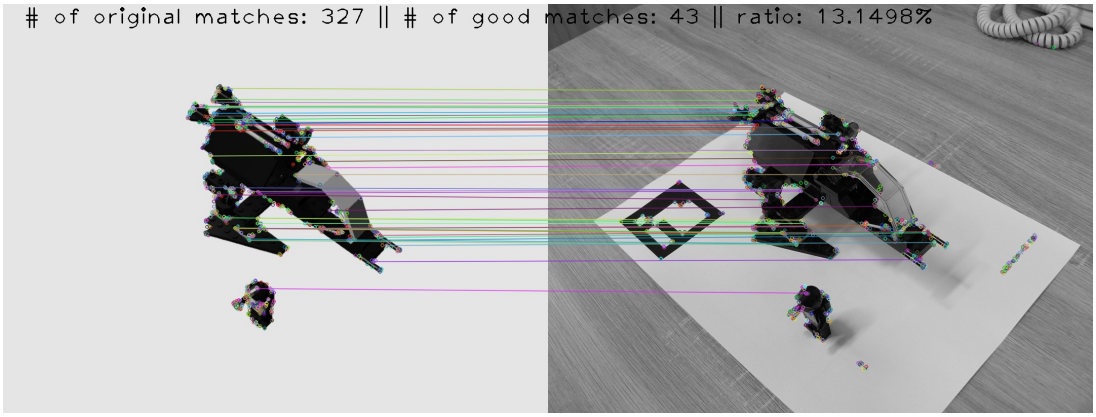
Obrázek 12: Vizuální výsledek algoritmu AKAZE.



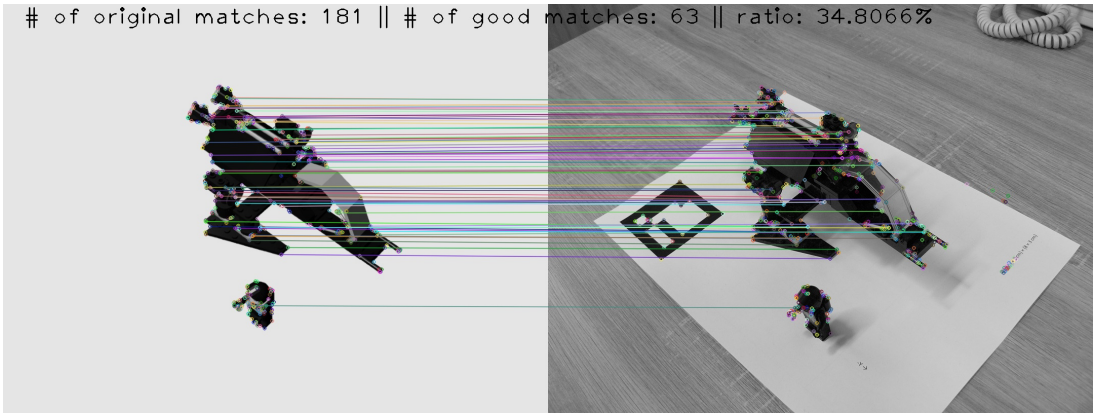
Obrázek 13: Vizuální výsledek algoritmu SIFT.



Obrázek 14: Vizuální výsledek algoritmu SURF.



Obrázek 15: Vizuální výsledek algoritmu ORB.

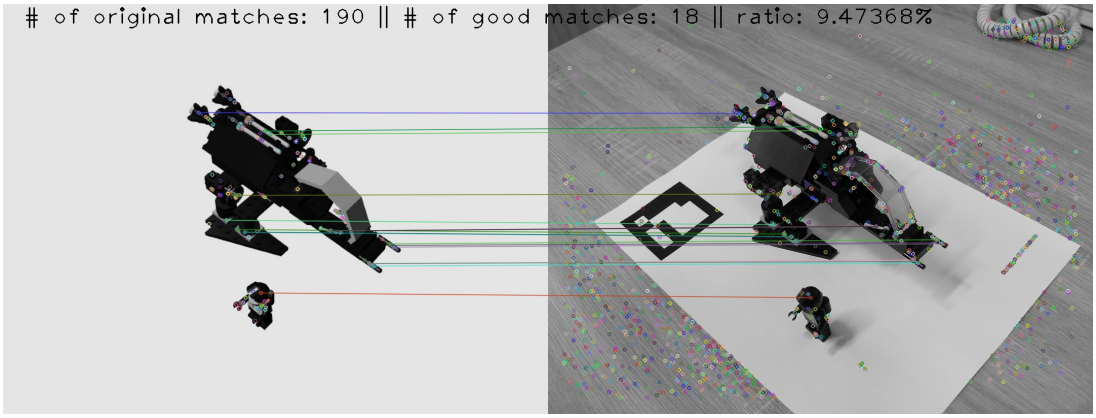


Obrázek 16: Vizuální výsledek algoritmu BRISK.

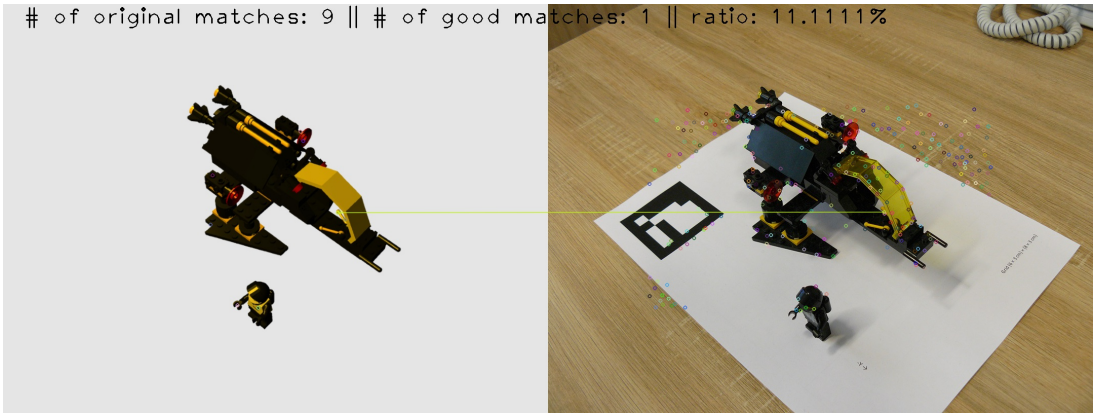
Obrázek 17: Vizuální výsledek algoritmu FAST.

Obrázek 18: Vizuální výsledek algoritmu FREAK.

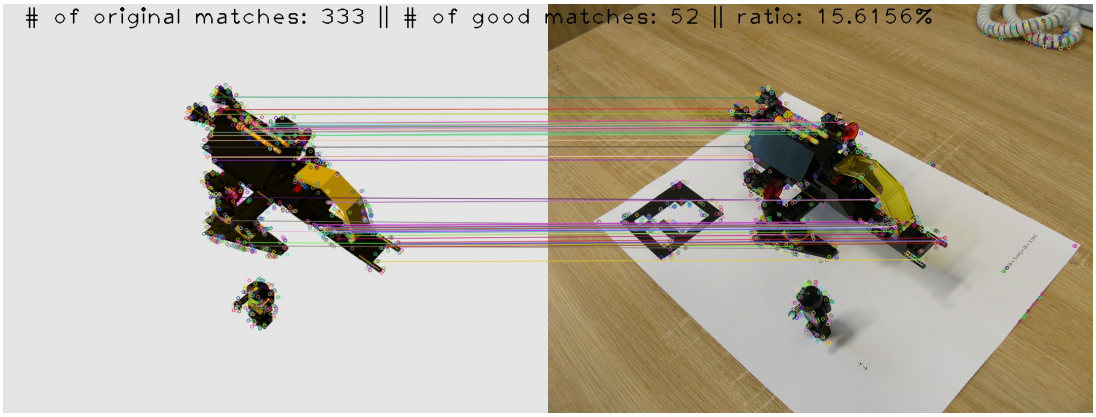
Obrázek 19: Vizuální výsledek algoritmu GFTT.



Obrázek 20: Vizuální výsledek algoritmu MSER.

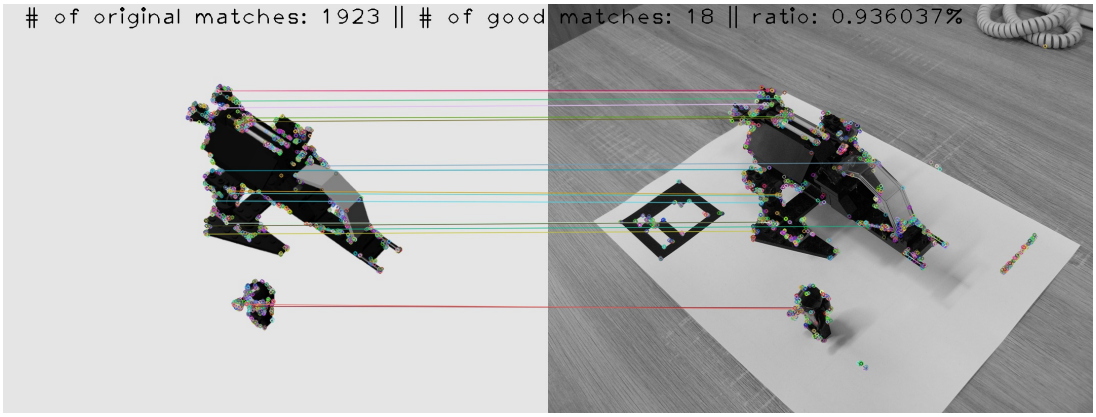


Obrázek 21: Vizuální výsledek algoritmu SimpleBlob.

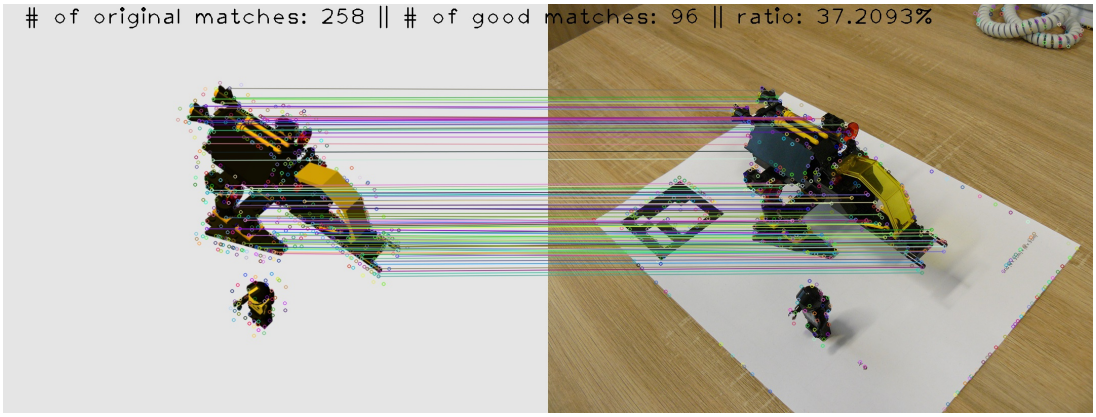


Obrázek 22: Vizuální výsledek algoritmu STAR.

Obrázek 23: Vizuální výsledek algoritmu BoostDesc.



Obrázek 24: Vizuální výsledek algoritmu ULATCH.



Obrázek 25: Vizuální výsledek algoritmu TILDE.

Literatura

[1] LEVI, Gil a Tal HASSNER. LATCH: Learned Arrangements of Three Patch Codes. Ithaca, 2016.

[2] VERDIE, Yannick, Kwang Moo YI, Pascal FUA a Vincent LEPETIT. TILDE: A Temporally Invariant Learned DEtector [online]. Boston, Massachusetts, USA, 2015 [cit. 2018-04-23]. Dostupné z: https://cvlab.epfl.ch/research/tilde

[3] KARIS, Brian. Real Shading in Unreal Engine 4 [online]. Siggraph, 2013 [cit. 2018-04-23]. Dostupné z: https://cdn2.unrealengine.com/Resources/files/2013SiggraphPresentationsNotes-26915738.pdf. Epic Games.

[4] FUJITA, Syoyo. Tinyobjloader [online]. [cit. 2018-04-23]. Dostupné z: https://github.com/syoyo/tinyobjloader

[5] STBI library [online]. [cit. 2018-04-23]. Dostupné z: https://github.com/nothings/stb

[6] Khronos Group, Inc. [online]. [cit. 2018-04-23]. Dostupné z: https://www.khronos.org/about/

[7] Khronos Group, Inc. [online]. [cit. 2018-04-23]. Dostupné z: https://www.khronos.org/members/list

[8] Khronos Group, Inc. [online]. [cit. 2018-04-23]. Dostupné z: https://www.khronos.org/news/press/khronos-group-releases-vulkan-1-1

[9] LAPINSKI, Pawel. Vulkan Cookbook: Work through recipes to unlock the full potential of the next generation graphics API—Vulkan. Packt Publishing, 2017. ISBN 9781786468154.

[10] WATT, Alan H. a Mark. WATT. Advanced animation and rendering techniques: theory and practice. Reading, Mass.: Addison-Wesley Pub., c1992. ISBN 978-0-201-54412-1.

[11] BLINN, James F. Models of Light Reflection for Computer Synthesized Pictures. New York, NY, USA, 1977.

[12] COOK, Robert L. a Kenneth E. TORRANCE. A Reflectance Model for Computer Graphics [online]. [cit. 2018-04-23]. Cornell University & Lucafilm Ltd.

[13] DEBEVEC, Paul. Image Based Lighting [online]. Siggraph, 2002 [cit. 2018-04-23]. Dostupné z: http://ict.usc.edu/pubs/Image-Based%20Lighting.pdf. USC Institute for Creative Technologies.

[14] OpenGL Mathematics [online]. [cit. 2018-04-23]. Dostupné z: https://glm.g-truc.net/0.9.9/index.html

[15] Oficiální webová stránka OpenCV [online]. [cit. 2018-04-23]. Dostupné z: https://opencv.org/about.html

[16] Nvidia memory management [online]. [cit. 2018-04-23]. Dostupné z: https://developer.nvidia.com/vulkan-memory-management

[17] RUBLEE, Ethan, Vincent RABAUD a Gary BRADSKI. ORB: an efficient alternative to SIFT or SURF. Menlo Park, 2011.

[18] ALCANTARILLA, P. F., A. BARTOLI a A. J. DAVISON. KAZE features. Fiorenze, 2012.

[19] LOWE, David G. Object Recognition from Local Scale-Invariant Features. Vancouver, 1999.

[20] BAY, Herbert, Andreas ESS, Tinne TUYTELAARS a Luc Van GOOL. Speeded-up Robust Features. Zurich, 2006.

[21] ROSTEN, Edward a Tom DRUMMOND. Machine learning for high-speed corner detection. 2006.

[22] CALONDER, Michale, Vincent LEPETIT, Christoph STRECHA a Pascal FUA. BRIEF: Binary Robust Independent Elementary Features. Lausanne, 2011.

[23] https://docs.opencv.org/3.0-beta/doc/py_tutorials/py_feature2d/py_orb/py_orb.html

[24] http://www.bmva.org/bmvc/1988/avc-88-023.pdf

[25] ALCANTARILLA, P. F., J. NUEVO a A. BARTOLI. Fast Explicit Diffusion for Accelerated Features in Nonlinear Scale Spaces. Bristol, 2013.

[26] LEUTENEGGER, Stefan, Margarita CHLI a Roland Y. SIEGWART. BRISK: Binary Robust Invariant Scalable Keypoints. Zurich, 2011.

[27] ALAHI, Alexandre, Raphaël ORTIZ a Pierre VANDERGHEYNST. FREAK: Fast Retina Keypoint. New York, 2012.

[28] SHI, Jianbo a C. TOMASI. Good features to track [online]. 1994 [cit. 2018-04-23]. Dostupné z: https://ieeexplore.ieee.org/document/323794/

[29] Simpleblob OpenCV [online]. [cit. 2018-04-23]. Dostupné z: https://docs.opencv.org/trunk/d0/d7a/classcv_1_1SimpleBlobDetector.html

[30] ROSIN, Paul L. a A. D. MARSHALL. BMVC 2002: proceedings of the 13th British Machine Vision Conference : 2nd-5th September 2002, Cardiff University. Great Britain: BMVA, c2002. ISBN 1-901725-19-7.

[31] TRULLS, Eduard, Kurt KONOLIGE a Morten Rufus BLAS. CenSurE: Center Surround Extremas for Realtime Feature Detection and Matching [online]. Marseille, 2008 [cit. 2018-04-23].

[32] YI, Kwang Moo, Pascal FUA, Vincent LEPETIT a Eduard TRULLS. LIFT: Learned Invariant Feature Transform [online]. Amsterdam, 2016 [cit. 2018-04-23].